MASTER IN COMPUTER SCIENCE

Electronic surveillance using Aibo for rescue teams

Gilson, Fabian; Struyven, Mathieu

Award date:
2005

University of Namur
Informatics Institute
School year 2004-2005

Electronic Surveillance using Aibo for Rescue Teams

Fabian Gilson Mathieu Struyven

Thesis provided in sight of getting
the Master Degree in Computer Science

Thanks

First, we would like to thank the whole **International Rescue System Institute**'s crew for their welcome and their help. All of them are highly responsible in our enjoyment in Japan. We will especially thank:

Professor Tadokoro who accepts us and supervises us during our whole stay. We will thank him offering us the opportunity to come in Japan and to work in his laboratory.

Masaki Minobe and **Konishi Kaoru** who supervise us for practical details related or not with the work as well as each student working at the lab for their warm welcome, their help and their availability for all our questions.

All the staff for their amiability and also their warm welcome, particularly **Kyoko** that helps us a lot for the administrative formalities.

Last, we especially want to thank our thesis guide, **Professor Schobbens** who help us to realize the present work.

Domo Arigato Gozaimasu!

Contents

List of Figures

List of Tables

Abstract

The objective of this paper is the use of Aibo, the Sony® robot, as a data collector for Rescue Teams via a web site. To complete this goal, a communication protocol was created and an intermediary server was needed to link Aibo and the site. This paper is divided by two, the first part contains a state of the art in Aibo programming and web technologies; and the second one explains how the three hosts are implemented.

L'objectif de ce papier est l'utilisation d'Aibo, le robot de Sony® en tant que collecteur d'information pour des équipes de sauveteurs via un site Internet. Pour atteindre ce but, un protocole de communication a été créé et un serveur intermédiaire était nécessaire afin de faire le lien entre Aibo et le site. Ce papier est divisé en deux partie, la première contient un état de l'art en programmation sur Aibo et en technologies Internet. La seconde partie explique comment sont implémentés les trois hôtes.

Chapter 1

Introduction

Current research in rescue systems heads for the use of robots. Since the eighties, many works focus on developing sneaking, running or flying machines remote-controlled or not by humans. These machines have two major goals. First, they search for survivors in catastrophe areas. Second, they limit the risks of casualties among rescuers.

The new craze for technologic devices offers a new opportunity for rescue teams. Entertainment robots like the Sony's dog Aibo possesses video camera, thermometer, infrared sensor and many more tools that could gather some information at any time. The project we developed uses Aibo as a data collector. The dog is connected to a web server where rescuers can ask it to take pictures or any other data it is able to collect. The underlying idea is the development of a communication protocol between any *process-capable* system in actual houses and a web site showing clearly all these collected data.

Along the development, we read a large scope of documentation for Aibo and for the web site. Aibo programming is essentially self-learning since the most part of the documentation is in Japanese. We hoped until the very last days an English version of the only available complete book on OPEN-R, the Aibo programming language. The OPEN-R web site contains enough source codes and specifications documents to understand more the language and the inner functioning of Aibo. The case is completely the opposite for the web server. We fell on the plenty of technologies and languages.

The present work is divided in two distinct parts. In the first part we makes a state of

the art of the approached technologies. The second explains our implementation of the communication protocol. Firstly, we talk about the research and development in robotics for rescue teams, the International Rescue System laboratory and the RoboCup events. Secondly, we describe the technical part of Aibo, its two programming languages and we go deeper in the OPEN-R one. Thirdly, we cross over the common web languages, highlighting their forces and weaknesses. Then, we do the same cross-analyzing for web servers and finally, we give a general view of the most used streaming protocols available.

The second part gives key comments over the implementation of the system. After a high-level analysis of the complete application, we explain the operating way of the communication protocol and its messages. Afterwards, the implementation details will be explained for the three collaborating hosts. We used an intermediate server to ensure the stability of the structure. We begin with this Local Server justifying our technology choices and explaining its main mechanisms. Then, we continue with Aibo. After a brief explanation of the chosen formalism to represent OPEN-R objects, we describe the four developed objects: *NetCom, Collect, ImageObserver* and *SoundRec*. The last host is the Web Server. We justify the chosen technology and language and we analyze the possibilities offered by the web site. Every host chapter is ended by a summary of the encountered problems. The last section regroups some improvements such as a simplification of the general architecture and security advices. This finishes with a high-level analysis of implementing streaming to complete the collecting possibilities.

Part I

State of the art

Chapter 2

Rescue system

This chapter presents human and robotic cooperation to help rescue operation in natural disaster. The first introductive sections describe the robotic's evolution and how some robots can help human in rescue operation. A second section presents different researches and events which takes an important part in rescue robot development. Finally, this chapter describes IRS, the Japanese laboratories where we made our research. Its scientific project and our role are described in this chapter.

2.1 Introduction

Following a disaster, many people gather to analyze the stricken area and to step in rescue operation. Those people include firemen, policemen, ambulance men, doctors, engineers or just voluntary people. Their first mission is to help humans buried in the damages but still alive, but it is not easy to performed. The area is often damaged or in fire, the rescue team has difficulty to progress and to find victims, but they do what they can to save, sometimes at the risk of their own life. So, they progress in the unknown and the layout of the place does not allow them to explore easily all nooks. They are sometimes blocked after a collapse, or encircled by the flames, and they do not know where the victims can be. Special intervention teams, like CRASAR[1] have been created

[1][33]The Center for Robot-Assisted Search and Rescue is a non-Profit Center at the University of South Florida, with Prof. Robin Murphy as the overall director. *"Our mission is to research new technologies for Search and Rescue, deploy these technologies as part of our international response support*

for big disasters, but in spite of their good preparation, they are often powerless in front of the tragedy.

In this context, international research strives to develop robots which can help in rescue operations. A robot can enter small paths too narrow for a human, some robots can support a higher temperature than human, a robot can communicate information to other robots or to humans with a wireless technology, so a robot can be a good help for rescuer team in some crisis situations.

2.2 Research in robotic rescue

2.2.1 Robotics overview

A robot is a machine able to execute tasks and manipulate objects according to a program. It has a physical structure appropriate for its environment (with sensors, captors...), a control program and can react differently according to circumstances, respecting its program. So the robot has entry ports to receive information from the captors and exit ports to transmit information to its "members motors" in view of accomplishing its tasks.

Robotics is the set of the studies and the techniques making it possible to work out robots. Designed to carry out tasks in place of a human, robots are more than simple computers: they must be able to perceive what surrounds them and to react consequently.

Intelligent system integrated in machine are tools which bring the computer technology power in every day life and commercial activity. Together, these technologies imitate and upgrade human capacity to perceive, reason, take decision and operate.

From 1980, these systems are present and can be found in many application domain.

Industry

According to Canadian research[40], intelligent systems have been first implemented for manufacturing, surface mine exploitation, forest, energetic production and in the

team, use our expertise as both scientists and field experts to evaluate rescue systems, and to train both emergency responder and other scientists on rescue robotics".

agribusiness sector. Robots were used to increase production, decrease costs and improve quality. The motor assembly industry has played an important role in the utilization development of robots in manufacturing sector.

Medecine[5]

Generally, surgery integrates robotics to interactively assist surgeon in planning and execution of surgical procedure. This clinical objective is to reinforce treatment quality limiting operating traumatism (reduction of incision size, tissue dilapidation, etc.) for patient and society benefits according to medical ethics. Concerned domains are neurosurgery, orthopaedics, microsurgery, laparoscopy, general and cardiac surgery.

Aeronautics and space

The idea in this domain is to develop robots which assist during flight and explore the ground in far planets. Researches and projects are led in many country like France, United States of America, Canada, etc. to develop exploring robots. Advantage of robots in this domain is a question of risk prevention. Robots can move easier than human in unknown ground, like on the moon, can enter some tight cavity and make pioneer work in exploration, and mostly, it can execute operation too risky for human.

The Rosa project[53], leaded by Canadian Space Agency and other independent spacial company is one of those projects. It aims to develop spacial robotic operating commands according to different autonomy level from total human control to total robot control. This project marries artificial intelligence technology, via computer vision and automated method of behaviour control to assist vision during space flight.

In USA, government and NASA authorize private company to make research in spacial robotic in view of develop a robot for spacial exploration (moon, march, ...) and assisted flight. The objective is to develop it for 2015 and is names Bush project.

Domestic

People have always dreamed that a machine could make house task in place of him. Today, with robotic, this dream is became a reality, people can find machine in store. Example are lawn mower and hoover. These technology progress completely autonomous

to accomplish its tasks. You can let the lawn mower running in the garden without surveillance, it will continually cut grass thanks to mark point which gives it ground coordinate. When it is powerless, the robot goes itself to its base battery. It is the same principle for hoover.

It exists also amusement robots that go with human like a domestic companion. The most well-known entertainment robot is the Sony's dog Aibo [47], which is described in the next chapter.

Another part of robotic usage concerns the research. The following section is devoted to explain this part of robotic.

2.2.2 robotic rescue

Research

University and laboratories made research to improve robotic performance with objective to make progress Science. Each year, an international general project, named RoboCup[2], is organized to develop artificial intelligence, robotic and relative. It is an attempt to stimulate artificial intelligence and clever robotics with a standard problem giving a large panel of technologies which can be integrated and examined. Robocup has chosen robotic soccer as a federative goal, but it is hoped that the research developed there will find applications in many industrial fields. The final aim of this project is : *[54] by the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team.* This objective is hard but it will bring a lot in robotic research. Many leagues have been founded in many country to join the competition : Laval[57], Cornell[56], Canuck[63], etc.

As explained in a previous section, robots can help human in rescue operation. An important part of robotic research is devoted to disaster mitigation.

A first step in the robotic rescue operation was to producing a preliminary model of how the search task could be conducted with robots and setting the requirements for robots.

CRASAR[33] has a central role in robotic rescue research. Its current project aims

[2] 2005 edition in Osaka [55]

to develop an adaptable shoring for robot-assisted search and rescue. The long term goal is to develop a network of distributed shoring robots.Each robots would be in the trouble pile and be able to adapt to the shifts in the structure. This would allow the structure to remain stable, protecting rescue workers and reducing future victims.

The first known use of robots for urban search and rescue was in 2001 after the World Trade Center attacks. During this operation, robots were searching victims, searching for paths through the rubble that would be quicker to excavate, making structural inspection and detecting hazardous materials. It was decided to use robots because traditional search equipment could not enter a space too small for a human or search dog, or could not enter a place still in fire or posing major risk of structural collapse.

The artificial intelligence community took interest in rescue robot as a challenging and worthy application domain. In this view, a competition has been created too and is named RoboCup Rescue[50]. Some leagues have been created to participate to this international competition : 5rings from University of Portugal (Portugal), ARK from University of Windsor (Canada), Bam from The University of Isfahan (Iran), etc.

Robocup rescue[50]

As explained previously, rescue operation following a disaster is a major social problem. It sets many heterogeneous agents in a hostile environment. The intention of this competition is to promote research and development in this significant civil domain. It includes coordination of multi-agents work teams, physical robots for research and rescue, numerical personal assistant, standard simulation and help for decision systems, evaluation referential of rescue strategy and a robotic system, all integrated in a complete and coherent system.This competition is build on the success of RoboCup-Soccer and provides technical forum and serious evaluation to researcher and practitioner.

At this moment, two projects and two competitions exist, the simulation RoboCupRescue project with aim to provide urgency decision support by integration of disaster information, prediction and planning tools and a human interface; the Infrastructure and Robotic project with aim to study new infrastructure and robotic standards including robots, clever villages, etc. to human welfare; the Simulation RoboCupRescue championship and the Robots RoboCupRescue championship are research development for the

both projects. In future, the integration of these activities will permit the creation of international electronic rescue team.

Here is an historic of past organized event for the RoboCup Rescue:

- RoboCup 2000 demonstration, CANADA, Melbourne
- RoboCup Japan Open 2001, JAPAN, Fukuoka (April 2001)
- RoboFesta Kansai 2001, JAPAN, Osaka (July 2001)
- RoboCup World Cup 2001, USA, Seattle (Augustus 2001)
- RoboCup International Symposium 2001, USA, Seattle (Augustus 2001)
- RoboCupRescue Robot League 2002, JAPAN, Fukuoka
- RoboCup 2002, JAPAN, Fukuoka / Busan (June 2002)
- Open Rescue Robot League 2003, JAPAN, Niigata
- Open Rescue Simulation League 2003, JAPAN
- Rescue Robot League 2003, ITALY, Padua (July 2003)
- Rescue Simulation League 2003, Padova
- Open Rescue Robot League 2004, JAPAN, Osaka
- Open Rescue simulation League 2004, JAPAN, Osaka
- RoboCup 2004, PORTUGAL, Lisbon.
- RoboCup Rescue - Robot League Camp, ITALY, Roma (November 2004)
- RoboCup Rescue - German Open, GERMANY, Paderborn (April 2005)
- RoboCup Rescue - Us Open, USA, Atlanta (may 2005)
- RoboCup Rescue - World Championship and Symposium, JAPAN, Osaka (July 2005)

International Rescue System Institute (IRS)[41]

"International Rescue System Institute is the industry-government-academia-civilian research organization to advance and diffuse high technologies coping with disaster. IRS aims to contribute to build a safe society in which people can live without anxiety by cooperation of various organizations and human resources".

This Japanese institute is divided into two laboratories : Kawasaki lab and Kobe lab which receive us to perform this project. Both laboratories lead research activities which are carried out by some academic or business organizations concerned, such as Kobe University, Kyoto University and Osaka University.

Almost all researchers are involved in the Special Project for Japanese Earthquake Disaster Mitigation in Urban Areas, launched by MEXT, the Ministry of Education, Culture, Sports, Science and technology. This project aims at significant mitigation of the earthquake disaster damage on the scale of the Great Hanshin Earthquake (Kobe, 1995) in big city regions. In the vanguard of the project, IRS carries out research and development of robots, intelligent sensors, portable devices and human interfaces which could integrate and gauge the information in disaster confusion and make the best rescue efforts suited to the situation.

Professor Tadokoro manages IRS from start and has a big role in robot rescue system research. As visitor researcher, we were invited to join the laboratory's research.

We took part in two different projects, one real and another experimental. IRS creates its own robots for rescue operations.

Each robot is composed of captors, sensors, ambulatory system, vision and program which manage all these robot's parts. The software included in each robot, manages a set of agents which are connected together and constitute the robot's brain. The number of agents could be high and difficult to manage. So a human interface has been created to set the agent's network and see a map of connected agents. The first project that we have joined was to improve this human interface.

The second project was a new experimental project of IRS. As well as IRS own robots are developed especially to help in rescue operation, it exists commercial robots which can be used for research. These robots are existing and it is necessary to adapt project according to the robot's possibilities. With this aim, we had to develop a reduced rescue

system with AIBO, Sony's dog robot. This project is presented in the second part of this work.

Chapter 3

Sony AIBO and OPEN-R programming

In this chapter, AIBO, the Sony's entertainment robot used for our project, is described. First, there is a section about AIBO's story, objectives and its specification. It will explain how AIBO can be used like a platform to conduct research in robotics, and finally there is a section about OPEN-R and R-CODE, the Sony's packages to develop program for AIBO. The main characteristics of the package and the process to develop application will be explained.

3.1 Aibo

3.1.1 Introduction[42][43]

AIBO is the nickname of the Sony dog robot provided from a 1993 project named "entertainment robot". This name has two meanings; it could be "A.I." (Artificial Intelligence) "bo" (from robot), or "aibou", a Japanese word meaning partner. So, AIBO is a genuine robotic partner with a dog's appearance for many points.

A robotic partner has to run without computer, not permanently connected to an electric source and, it has to perceive and communicate with the world around it. In this way, Toshitada Doi, AIBO's creator, has developed his robot like a dog with emotions (happiness, sadness, anger, surprise and fear), instincts (affection, curiosity, practice,

hunger and sleep), perceptions (hearing, touch, sight and equilibrium) and communication (speak, play music, make gesture, make light).

These behaviours and feelings are defined in a software, residing on a removable "memory stick" , they are implemented thanks to captors and actuators. Sony provides various software which are available according to AIBO's release. This software makes it possible in particular AIBO to play with a ball, i.e. to recognize it and move towards the ball or to make acrobatics.

In June 1999, ERS-110 was the first AIBO commercialized in Japan and U.S.A. This first release was a Hajime Sorayama development, which gave it eighteen degrees of freedom. ERS-110 was replaced by ERS-111 which is more robust but, as its ancestor, built in limited edition. In November 2000, the first unlimited version of AIBO was built, the ERS-210. This one has shrill hear and smoother features. This release is considered to be the starting of AIBO's fashion. After these releases, Sony wanted to change its appearance with a more robotic and more metallic new body. So, the ERS-220 was created during 2001 with same technical characteristics than ERS-210. In 2002, ERS-210 and ERS-220 were upgraded to ERS-210A and ERS-220A. The processor clock speed was doubled from 192 to 384 MHz. Both releases are currently not commercially available.

The last AIBO built by Sony was the ERS-7 in 2003 which offers more degrees of freedom in the neck with a view to manipulate bones. This AIBO is able to return on its base when it feels discharged. This release is the only AIBO still on sale. To buy the former versions, it is necessary to turn to the second hand market.

3.1.2 Specification and interest in robotic research

Specification[52]

Each release of AIBO provides distinct specifications and availability concerning degree of freedom, CPU, memory, built-in sensors, power consumption, network connection, dimension, etc. Figure 3.1 provides a picture of AIBO ERS-210A, the AIBO model used in this project. This upgraded high-speed version of the ERS-210, has a new advanced central processing unit. It provides twenty degrees of freedom, one for mouth, three for

head, three for legs, one by ear and two for tail. It is composed of seven built-in sensors, a temperature sensor, infrared distance sensor, acceleration sensor pressure sensors (head, back, chin, paws), vibration sensor, a PC card slot Type2 In/Out, a Memory Stick slot In/Out and and AC IN power supply connector input. It is also constitute with audio output and built-in clock. The average power consumption is approximatively 9W and it permits an operating time of 1.5 hours during standard operation. It can stand temperature from 5 to 35 degrees Celsius (41 to 95 degrees Fahrenheit) and a humidity rate from 10% to 80%. This robot is a small robot, 6.06" weight * 10.47" high * 10.79" large, which weights approximatively 1.5 kilograms. Figure 3.1 provides a picture of AIBO ERS-210A with its pink ball, the AIBO model used in this project.

Figure 3.1: AIBO ERS-210A from Sony.

This model was the first with a wireless LAN card and so the first able to exchange information with other AIBO and a computer. This functionality made it possible to make true experiments with AIBO. Today, this model has been substituted by the ERS-7, so, it is not possible to buy a new one.

Interest in robotic research

Aside the numerous captors and actuators, another important advantage is that Aibo is programmable. So, AIBO is not only a partner with Sony's software, it can be adapted according to the owner desire. AIBO dan be used in robotic research thanks to technology platform provided by SONY. Two technologies are available to program AIBO: R-CODE SDK and OPEN-R SDK. These two technologies are by definition rather different, the first one is user-friendly and partially limited, while the second one provides a complete powerful library. Both SDKs[1] are freely downloadable on the AIBO SDE[2] web page, the official AIBO's website[52].

Next sections will present and compare both technologies. A first section will present OPEN-R SDK and a second one will present R-CODE SDK and compare it with OPEN-R SDK.

3.2 OPEN-R SDK

"OPEN-R" is the standard interface for the entertainment robot system that Sony is actively promoting. This interface greatly expands the capabilities of entertainment robots. The OPEN-R SDK is the cross development based on gcc (C++) where you can make software on AIBO. OPEN-R API permits to use AIBO's function such as move AIBO's joints, get information from sensors, get image from camera, use wireless LAN (TCP/IP).

The development environment provided is completely free of charge and includes tools to make OPEN-R objects, sample programs, and "Memory Stick" images, which can be put on a AIBO "Programming Memory Stick". It is possible to program with GNU Tools, like **gcc** or Cygwin. The software developed with this environment can be distributed freely, but commercial usage is not allowed.

There is some documentation and tutorials on the web site[3]. Those give a broad outline of technology and its components as well as councils of utilization. Besides, Sony

[1] Software Development Kit

[2] Software Development Environment

[3] The Official Website[52] provides document *OPEN_R_SDK-docE-1.1.5-r1.tar.gz* in download section

provides the "official" Programming Book[16], which explains how to use the configuration's files, the main packages and described in a detailed way some concrete samples OPEN-R programs which can turn on AIBO. Unfortunately, this book is currently only available in Japanese edition.

3.2.1 Generality

Characteristics

A *modularized hardware* gives the possibility to change the robot's form by exchanging modules (e.g. you can change a leg module or the head module). Each module is connected by a high speed serial bus featuring auto-detection of the robot's hardware configuration.

More, it offers *modularized software* modules called "objects". It is relatively easy to replace running objects, these ones are loaded from the *Memory Stick*. OPEN-R programs are built as a collection of concurrently running OPEN-R objects. A Modular program offers three advantages. Clarity of the design, each object handles a set of specific behaviours. For example an object could handle all the head movements like "looking ahead", "looking down" and "swinging the head" while an other could handle the legs like "raising to standing position", "walking forward" and "stop". A second advantage is the easiness of reusing pre-existing objects: they don't have to be compiled again since they communicate with passing message. And last but not least: a natural parallel processing.

OPEN-R offers yet a *networking support* with Wireless LAN supports and TCP/IP network protocol support.

Specification and advice

OPEN-R is a multi-platform environment (Windows 2000 Professional/XP, FreeBSD, Solaris, MacOS X and other UNIX-based) which require at least 233Mhz CPU and 64MB memory. It require at least 200MB free space and to be able to read and write to a "Memory Stick". The PC has to be equip with a IEEE802.11b compliant Wireless LAN card or IEEE802.11b compliant Access Point connected to the LAN.

Of course, an AIBO is needed. The following releases support OPEN-R : ERS-7, ERS-210, ERS-220, ERS-210A and ERS-220A. It is also necessary to have a AIBO "Programming Memory Stick" (ERA-MS008 or ERA-MS016) to put the developed software. Only this Memory Stick can be used with the OPEN-R SDK and it may not be formatted. A AIBO Wireless LAN Card ERA-201D1 is also needed. It is to note that ERS-7 has a wireless LAN function built-in. Someone who wants develop with OPEN-R needs to assume some knowledge:

- C++ programming or other object oriented languages
- GNU development tools **gcc/g++**, **ld** and **make**
- Cygwin tools
- Shells and Unix-like commands

The inside of the OPEN-R system layer API can be regarded as an "operating system" for robots. The OPEN-R SDK meets the demands of people who want to develop the essential part of robotics programs without being bothered by the complicated items inside the system layer. The following sections will give some basic knowledge and mechanism which ave to be known to develop with the OPEN-R SDK.

3.2.2 OPEN-R objects and Inter-Object Communication

As explained previously, OPEN-R programs are built as a collection of concurrently running OPEN-R objects. An OPEN-R object is implemented using a C++ object but these are two different concepts and should not be confused. Each so-called object runs concurrently with the others and objects are able to communicate with each other by passing message. Before to define what is an OPEN-R object, it is necessary to explain this mechanism, named Inter-Object Communication, which permits objects to communicate together.

The communication is made by a message passing protocol. A message can be a C++ primary type (int, float ...), an array, a structure, a class or a pointer. The message contains also a selector, which is an integer that specifies a task to be done by the

receiver of the message. When an object receive a message, the function corresponding to the selector is invoked, with the data in the message as its argument. As objects are single-threaded, it can process only one message at a time, other received messages are put into a message queue and processed later.

In this protocol the notion of communication axis is introduced. A communication axis is composed by unidirectional communication channels between two objects, the *subject* (which sends data) and the *observer* (which receives data).

Each channel has one fixed subject and one fixed observer. So two channels are required at least to make bidirectional communication axis. Only one type of message can go through a channel. In the OPEN-R SDK, only channels exist, there is no implementation of the notion of communication axis, which has been presented for sake of clarity. So, each subject and observer has to define its gate, with a name, a type, and selector, to create the channel. These are defined in a configuration file (see section 3.2.3).

This protocol is a synchronization protocol because both objects exchange messages to know if the other one is are ready to send or receive data. Figure 3.2 pretends to describe the mechanism.

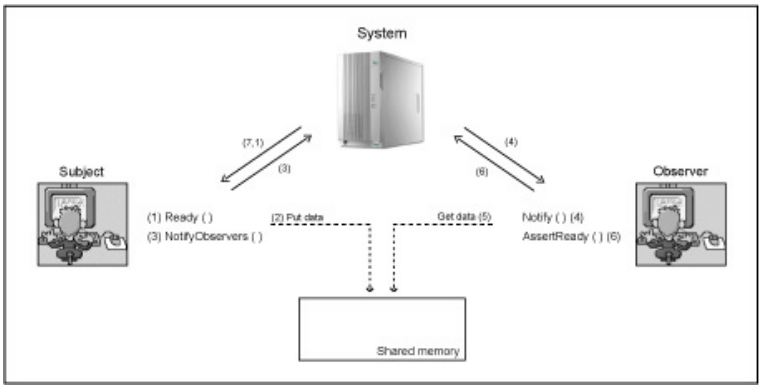

Figure 3.2: A synchronized protocol to exchange data between OPEN-R objects.

The subject has the function `Ready()` to send data to the observer. When it sends data, this one is put in a public area, a shared memory and the function `NotifyObservers()` is called by the subject to inform the observer that the data is sent.

The observer has the function `Notify()` to receive data. This function is invoked by system after subject has called `NotifyObserver()`.The observer can then get the data

in the shared memory and process it. To prevent the subject that it is ready to receive other data, it calls the function `AssertReady()`.

So, An OPEN-R object is compiled from a C++ class. Here is the typical life cycle of an objet :

- Loaded by the system
- Wait for a message
- When a message arrives, execute the method corresponding to the selector specified in the message. Possibly send some messages to other objects
- When the method finishes, wait for a message, etc.

At run-time, objects are represented by C++ class named core class. Each object should be represented by only one core class. Objects have entry points for receiving messages (as explained before). A core class must inherit from the *OOBject* base class. This inheritance relation requires the implementation of four virtual functions : `DoInit()`, `DoStart()`, `DoStop()` and `DoDestroy()`.

`DoInit()` is called when the object is loaded by the system. This function initializes the gates and registers subjects and observers the objet communicate with. Macros are predefined in the OPEN-R SDK to simplify this initialization requirements.

`DoStart()` is called after `DoInit()` is executed in all objects. Here, each observer usually sends ASSERT-READY to its connecting subjects.

`DoStop()` is called at shutdown of the system. This function usually stops outgoing gates and sends message to prevent it to all his observers. This means that this object cannot receive a message anymore.

`DoDestroy()` is called at shutdown after `DoStop()` has been called on all objects.

Each object has a state which has to be defined every time in the running time. This state represents the behaviour of the object, its reaction on message incoming Thus, it is careful to initialize the state in the object's constructor (IDLE state, for sample).

3.2.3 Building software

There are many steps in a program flow of development. From design of the objects until debugging, each step has to be filled carefully. First, the development consist to create objects and configuration file required by OPEN-R. This step begins by the design of object consisting to design the functions of the object which is going to create, as well as the flow of data between these objects.

Next, design the data type for inter-object communication. This step permits to create the *stub.cfg* which describes the connection between objects and the entry points relatives to the communication channel. This file will be read by the OPEN-R SDK tool named Stubgen2, which creates files defining several macros and constants, just before calling **gcc**. It has to be created even if there is no communication with other object.

This configuration file created, the core class can be implemented. This core class includes the functions specified by stub.cfg, the four virtual function inherited by *OObject*, and other member functions.

Next, the configuration of objects at run-time are specified in the *objectName.ocf* (where *objectName* is the object's name which is developed). This file contains useful information used when linking the libraries.

Is is also advised to create a *makefile* for each object, to set all OPEN-R SDK tools commands. Each object's files (*.cc, *.h, stub.cfg, *.ocf and *makefile*) are put in a directory identified by the object's name. Each object's directory are put in a same parent directory, with a make file and a "system" directory named MS (for Memory Stick). For sample, if a project is designed with one object called *myObject*, it could have a structure like this :

../project/
../project/makefile
../project/MS/
../project/myObject/
../project/myObject/myObject.cc
../project/myObject/myObject.h
../project/myObject/stub.cfg
../project/myObject/myObject.ocf

../project/myObject/makefile

The next step is to build the executable file for created objects by linking with other necessary libraries (specified in the *makefile*). Objects are now completely created, but it remains some configuration setting. The MS directory will contains the executable file which will be put on the Memory Stick. This directory contains also three setting files which will be needed at run-time. The *Object.cfg* describes objects to execute, the *Connect.cfg* is a summary of connections between objects and the *Designdb.cfg* describes some files, with paths, which are accessed by objects at run-time.

The MS directory contains the OPEN-R directory which can be copied on a AIBO Programming Memory Stick. There are two steps to run OPEN-R programs on the AIBO's Memory Stick. First the base system must be installed on the Memory Stick, and then, on top of this, copy the OPEN-R directory contented in MS directory, with the objects created. Three configurations on the base system are available. The *Basic*, without LAN environment, the *Wlan* with a wireless environment but without console and the *Wconsole* with wireless environment and console. The Memory Stick loaded can be put in AIBO to be executed.

3.2.4 OPEN-R over network

This part describes the protocols in the current version of the Ipv4 protocol stack and explains how objects communicate with the stack in view to reach a remote host. IP version 4 (Ipv4) is currently the most widely used version of this protocol, and is the version available on OPEN-R. The Ipv4 protocol stack on OPEN-R includes several protocols that supplement the basic IP protocol. TCP[4], the protocol of our choice (reason is explained later in the work) is present in Ipv4. it runs on top of IP and provides objects with a connection-oriented, reliable and byte stream service. Ipv4 contains also IP, UDP[5], DNS[6] and DHCP[7] protocols. The network parameter are defined in the configuration file described previously.

[4]Transport Control Protocol
[5]User Datagram Protocol
[6]Domain Name System
[7]Dynamic Host Configuration Protocol

On OPEN-R, the IPv4 protocol stack is implemented using the OPEN-R Networking Toolkit (ANT). The stack exists at runtime in the IPStack, which also includes the ANT runtime environment. This IPStack is an OPEN-R system layer object which provides Networking services on OPEN-R. Objects communicate with the protocol stack through normal message passing, by sending special messages to and receiving special message from the IPStack.

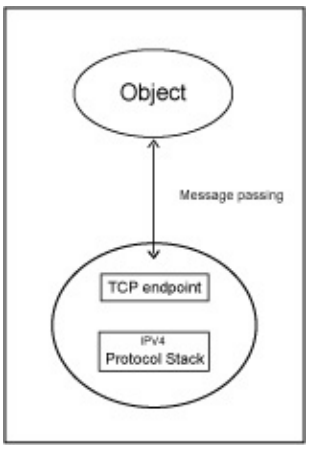

Figure 3.3: IP figure.

The first thing than an object must do is ask the IPStack to create an endpoint. An endpoint is a special ANT construct that is located at the top of the protocol stack and is responsible for communication between the object and the stack. An object requires one endpoint per Network connection.

In addition to its endpoints, an object must also create one or more shared memory buffers. These buffers are required for exchanging data between objects and the IPv4 protocol stack. Shared memory buffers map a common memory area into the address space of both objects. This is necessary because both objects share not necessarily the same address space and cannot exchange pointers to the data being transferred.

All messages sent to the IPStack are inherited from the *antEnvMsg* structure. This structure provides the basic message handling constructs such as `Send()` and `Call()`. Each service offered by a protocol has a specific inherited message. For sample, a request to send data by TCP is made with a *TCPEndPointSendMsg*. In our application, we use the followed service: `Listen(), Send(), Receive()` and `Close()`.

The explanation of the four next objects is based on the *finite state automaton* defined in section 9.1. This four respect the formalism describe in the general overview.

3.2.5 System objects

It was explained in a previous section that objects could communicate together. Like this, objects can also communicated with objects interfaces with Aibo's hardware. Each sensors and captors can be asked by an object via system's object provided by Sony. Objects that interface with Aibo's hardware have the same behaviour than all other OPEN-R object.

OvirtualRobotComm interfaces with all Aibo's joints, sensors, LEDs and camera. *OvirtualRobotAudioComm* interfaces with Aibo's audio device.

These objects use different messages types to communicate. *OcommandVectorData* is a data structure that holds joints and LED commands, *OsensorVectorData* is a data structure that holds sensor information, *OfbkImageVectorData* is a data structure that holds image data and *OsoundVectorData* is a data structure that holds sound data.

Next to these objects that interface with the hardware, another system object is interesting and recommended to develop a project with Aibo: *PowerMonitor*. This object monitors the battery state and manages the shutdown action.

3.3 R-CODE SDK and comparison with OPEN-R SDK

R-CODE-SDK offers an environment to execute a simplified interpreted scripting language that can be used to program AIBO ERS-7 (only, unfortunately). This environment permit to use sensor data, variables, R-CODE's built-in commands and more to program AIBO.

The R-CODE SDK is suitable for beginners, because it is very easy to create robot programs. With R-CODE, you can use 600 built-in motions that are included in commercial AIBOware, and which is more, its commercial usage is allowed.

R-CODE SDK can runs over any OS, CPU and memory. It requires at least 20MB free space and that computer can read and write on a Memory Stick. A IEEE802.11b compliant Wireless LAN card can be useful for debugging and make the program com-

municate with a remote computer.

As this environment is not compatible with AIBO ERS-210A and gives not a total control on AIBO, we have chosen to use the OPEN-R SDK to develop our application. In view to convince the reader, here is some comparison between the both environments presented before.

OPEN-R package permits to have a total control on AIBO. You can make software that works on AIBO (ERS-7, ERS-210, ERS-220, ERS-210A, and ERS-220A). You can search robotics programming such as PID control and experiment with new walk styles; you can make a program that exploits AIBO's hardware limits, such as the programs in Robocop, but commercial use is not allowed. The OPEN-R SDK creates native MIPS binaries that run on AIBO, what's useful for applications that require quick responses.

R-CODE SDK is an environment where you can execute R-CODE scripts/programs on AIBO ERS-7 only. You can have AIBO walk or dance with only a few lines of code. It is not compiled because it is a scripting language. As it is an interpreted language, it is not recommended for complex calculations, but it is easy to modify the script program and re-run.

So, the Sony Aibo robot is currently a very interesting platform to conduct research in Robotics and Artificial Intelligence.

Chapter 4

Common web languages and technologies

This chapter will cross over the most common web languages. We will try here to highlight their forces and weaknesses and at last, we will compare them following some important topics.

4.1 Server-side versus client-side processes

In the early days of the Internet, every page was directly written in *HyperText Markup Language* (HTML), this means that they all displayed static content. Indeed, HTML is a tag-based language and it does not provide any tool to process information and then display a web page using this information. It simply means that web pages were always the same for each visitor.

In these early days, web sites were only used to display static information, but quickly, many languages were invented to process information in the background and then generate HTML code related with the inputs. For that, there are two ways to process data: on the server-side and on the client-side. On the first hand, on the client-side means that the user web browser makes some work on the web page, for example it could check if a line of an application form is well filled. On the other hand, server-side languages are generally more powerful than client-side languages. They could generate

on-the-fly content depending on, for example, the user or on the information available at that time. This permits more flexibility and more closeness to the website.

In the following, we will cross through these major languages and then try to highlight some important assets of these languages.

4.2 Practical Extraction and Report Language (Perl)

4.2.1 History and overview

Perl is born in 1987, made by Larry Wall, a UNIX programmer. It arrived in the emerging Internet requirements. At that time, L. Wall was developing tools to extend the Awk[1] shell command language for efficient text manipulation. Perl is a high-level general-purpose programming language originally developed for text manipulation. Now, it is used for a wide range of topics like web development, network programming, GUI[2] building and many more. Perl is more practical and easy to use than beautiful and elegant, according to its inventor. It supports easily procedural and object oriented programming and has one of the world's most imposing collections of third-party modules.

4.2.2 The syntax and power of Perl

Perl is a C-like scripting language designed for efficient text processing. It is really useful for writing web applications. It provides also a large panel of libraries for database management, network programming (with *sockets*) and also GUI[3] programming. That is why it was widely used in the beginning of the Internet. Perl is really similar with languages like C, shell programming or its *father* AWK, so it is easy to learn for programmers used to write code in these languages. One of the most interesting assets of Perl is the possibility to link in scripts some application codes written in C or C++.

Perl is a loosely typed language as it knows three types: `scalars`, `arrays` and `hashes`. `Scalars` are single variables, `arrays` are indexed lists of possibly non homogeneous `scalars`, and `hashes` are sets of pairs of the form {*key, value*}.

[1] Awk is a powerful text-processing language by Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger launched in 1988.

[2] Graphical User Interface

[3] Graphical User Interface

CGI scripts are highly portable since it is a very hold standard and now all web servers support them. Many libraries are freely available for a large range of tasks, but *all-in-one* packages begin to be hard to find.

Lastly, Perl is an Open Source language only supported by volunteers and totally free. After the syntax definition and the first interpreter releases, many people associated themselves to the project and grew up the Perl development community. Since the first release, a huge number of tools and libraries were added and improved for instance operations over regular expressions.

4.3 Microsoft's Technologies

4.3.1 Active Server Pages (ASP)

Active Server Pages is a technology (not a language) introduced by Microsoft in its Internet Information Server 2.0 (IIS 2.0). The most common scripting languages for ASP are VBScript and JScript. The first is a server-side scripting language, subset of the Microsoft's Visual Basic, where its name comes from. The second implements the same standard as Netscape's JavaScript (ECMA-262), so is client-side.

VBScript is a loosely typed scripting language. There is only one type of variable called `Dim` but there are many subtypes like integer, bytes, long integer and some structures too. But, like many scripting languages, the typing system is a bit flabby. It provides a large execution and syntax error handling with meaningful codes. This permits a good error control without appealing to the classic `print(DEBUG)`. It permits also to define object oriented classes with their own variables and functions.

The power of ASP consists in its ActiveX components (ADO[4]). ActiveX is a loosely defined set of technologies developed by Microsoft for sharing information among different applications. Concretely, ADO are object oriented structures permitting to add to a web page anything like sounds, animations, animated menus or Java applets. But, these components are only executable on hosts running Windows and surfing with Internet Explorer. Furthermore, ASP pages only run over Windows platforms.

[4]ActiveX Data Object

4.3.2 The brand new .Net technology

Microsoft launched by the year 2002 its new .Net framework. It consists in a global revision of the development options of Windows. The execution platform was unified and web services were placed in the core of the framework. .Net is now an optional component of Windows but soon, it will be probably integrated in the kernel.

.Net is principally object oriented but supports the non object oriented construction of some languages. It offers a *common approach* for independent languages as Perl, VBScript, C++ and C#. It defines in fact a **common cross-language strongly typed standard** which eases the coexistence of different languages in the same application. The code is compiled (using the inherent compiler) in an *object code* completely platform independent. Furthermore, .Net applications are known as *managed code*. Garbage collecting, interoperability problems and cross-language debugging are controlled by the system. The fundamental notion consists in *assemblies*. It is nothing else than packages like Java ones with their variables, typing and security scopes. Last but not least, different versions of the same package can run in parallel.

Unfortunately, because this new technology uses a strongly typed language, not only old DLLs[5] have some compatibility problems, but also the Microsoft's ODBC[6], ADO and OLE-DB[7] are deprecated in favor of ADO.Net. And last, .Net's strongly typing makes hard to use ActiveX objects, deprecated in favor of the .Net extension letting us thinking that ASP (moving to ASP.NET like ADO moved to ADO.Net) will probably disappear in the future.

4.3.3 A few words about C#

In spite of its name and its origin, C# (pronounced C-Sharp) looks more like Java than C. It is a combination of C, C++ and Visual Basic, but as in Java, the programmer never sees any pointer except in defined *unsafe code*. It is a pure object oriented languages with GUI development facilities and the usual OO concepts. Except some purely subjective improvements and typing haggling, C# and Java are the same, but is very effective

[5]Dynamic Link Library: function libraries on Microsoft environment.
[6]Open DataBase Connectivity: a set of database control programs.
[7]Object Linking and Embedding-DataBase: same as ODBC

coupled with the .Net technology. Microsoft just write recently an ECMA standard to fix the language specifications in a bulky report.

4.4 Home Page Hypertext Preprocessor (PHP)

4.4.1 Birth of a free technology

PHP is an incredible success story. It was originally designed as a scripting language for adding some basic dynamic content on a web page like the famous visitors' meter, but grew rapidly. Written in Perl at the beginning, it has been translated to C. PHP is constructed more simply than Perl or C, so is by the way less powerful. However it permits a very simple control and interaction with databases and is complete enough for a large number of web application. Since its first release in 1994 from the hand of Rasmus Lerdorf, an American programmer, PHP enlarged its action scope. It begins to be widely used on the net nowadays and it imposes itself as the most dangerous competitor of Windows' ASP environment.

4.4.2 Syntax of PHP

At the beginning, PHP was not an object oriented language but some OO features were added with time. The syntax is simple, PHP code is intermingled into the HTML code using predefined tags where some dynamic processing is necessary. The syntax is exactly the same as the C/C++ syntax for loop, test and operations over variables. At the opposite of many scripting languages, PHP is strongly typed. Last, *handling* and *throwing* of errors are not available yet, so the programmer has to do them by himself if he wants some.

4.4.3 A multi-platform language

PHP is an Open Source language and is completely free. It is also supported on Linux as well as on Windows environment. More and more web servers worldwide are accepting PHP and it is completely supported by every web browser because it is a fully server-side

language as JSP[8].

Furthermore, because it is designed simply, pages interpreting are not a heavy task for web servers. It has been tested to support very loaded servers and it gives a good answer to all security problems of the Microsoft's languages. It can be run over Apache, IIS and a lot of other web severs, it fully and simply supports all database types and there exist a lot of "beginner packages" with all what you need to write and run your first PHP web page.

4.5 Macromedia ColdFusion

4.5.1 Overview

ColdFusion was originally created by the firm Allaire which was bought by Macromedia in 2001. It is presented as a fast reliable way to construct and deploy scalable web applications. It supports the most usual databases like Microsoft Access, SQL or Oracle.

4.5.2 WYSIWYG approach

To write your web pages, you have the choice between writing code by yourself or using a drag-and-drop interface and asking your development tool to generate the corresponding code for you. This environment eases designing web pages and permits beginners to make their first web site very quickly and easily. For developers wanting to write the code by themselves, they have to learn the ColdFusion syntax which is almost like C, but, the manufacturer insists on the *WYSIWYG*[9] side of its environment.

4.5.3 Syntax of ColdFusion

ColdFusion place itself in the same slot as JSP and PHP since it is a tag-based scripting language. But it strongly works like the Microsoft ASP technology, especially the VBScript language. It is a loosely typed language and you need the complete environment proposed by the language manufacturer. Your code is not portable to any other concurrent web server (running for the most part on Windows), but you are sure that

[8] see section 4.6

[9] What You See Is What You Get

your solution will perfectly work if the code is well written because language and server are hardly interdependent.

Since the release 6.0, object oriented features called ColdFusion Components (CFCs) were added. These objects can be compared to ADOs since their inner and outer functioning are similar. Also as ASP, this technology is really expensive and its development tools (or servers) are available only on Windows and MacOs every platform.

4.6 JavaServer Pages and Servlets

4.6.1 Overview

JavaServer Pages and Servlets were created by Sun Microsystems in the middle of 1999. At that time, many companies had already developed their languages, remember that PHP was released in 1994. But, as said before, all processes written in these languages are interpreted scripts that provide no debugging facilities. Sun wanted to use the mighty debugging tools and the error-handling facilities of Java to create its own web language.

4.6.2 Power of Java

Servlets are real Java classes with two special method *doGet* to hand HTTP GET requests and *doPost* for POST requests. But, in order to generate the HTML code, servlets' developers have to write a lot of *out.println()* statements containing the HTML code as answer.

JavaServer Pages are HTML files with special tags used to insert some Java code in it (as PHP). It provides many defined built-in facilities to ease the developer's job. When a web page written in JSP is requested for the first time, the Java compiler translates the JSP file into a servlet. So the two methods create the same result, but the JSP syntactic sugar is useful when the dynamic content is small.

There exist some development environments which ease the creation and the deployment of Java-based web applications. Usually, these tools create every needed configuration files by themselves and they help for files' headers and the syntax.

4.6.3 Syntax of JSP and Servlets

On one hand, the syntax of JSP code is very simple as it is a tag-based language using Java, but the very difference of JSP compared with other tag-based languages it that JSP files are compiled into servlets. Moreover, as in PHP, there are many built-in facilities for including files, database connections, etc. On the other hand, the syntax of Servlets is just Java code with associated packages. Developers have to rewrite two methods *doGet* and *doPost*, and they have the possibility to make some work during the initialization of the servlets.

Java is an object oriented language, which is a very profitable advantage within the context of web applications development and that both use the Java syntax, so variables are strongly typed. Java is also known as completely platform independent and the SDK[10] is totally free of charge even if the Java language is not an Open Source technology. Finally, the Java Community counts a huge number of developers and many additional independent packages are spread all over the Internet.

4.7 Tool Control Language/ToolKit

4.7.1 The functional approach

The last language we will see is a multi-paradigm one. But, what interests us is the functional coding possibilities of Tcl. This kind of approach is very useful because tag-based code is easily generated and well balanced by this method. Tree-like structure are easily processed in a recursive way and functional programs or scripts are more efficient since as they process only needed data and they make it using the fastest way. But, because of their power, they seem extremely cryptic at first sight and they are known as difficult languages to learn.

4.7.2 Mighty and easy

Tcl/Tk (pronounced "tickle tee kay") is a general-purpose command/ scripting language (like Perl) from the hand of an American named John Ousterhout. In 1988, he was

[10]Software Development Kit: complete environment with compiler, packages, etc. for developers.

working with its students on Sprite, a network operating system and Tcl was created as the command language. An extension named Tk was added later consisting in a *ToolKit* of widgets (for *window gadgets*), which are graphical objects. The toolkit contains a huge amount of objects such as buttons, checkboxes or item lists to create and create easily GUI (web pages or softwares' user interfaces).

Tcl/Tk is very powerful for a large variety of jobs. It provides by example many functions for socket programming and a powerful Regular Expression processing. The development kit contains also a test tool as well as a stack recuperation system to ease debugging.

4.7.3 Syntax of Tcl/Tk

The syntax always looks like `function args*`[11]. There exists of course basic instructions as loops, tests, assignments, etc., and, as said before, many general-purposed functions.

Tcl is one of the weakly typed language. As variables, it knows (non indexed) `lists` (that could contain other lists), (indexed) `arrays` and `strings`. The Tk extension adds the widgets which could be buttons, checkboxes, item lists, etc, for quick and easy GUI design.

Tcl/Tk code can run on almost every UNIX platform, on Macintosh and on Windows too. It is also possible to embed C/C++ commands by multiple ways. But above all, everybody can easily write extension of parts of the Tcl language and, once again, Tcl is another Open Source product.

4.8 Other languages

It is impossible to talk here about every languages that exist. We mentioned previously Awk, but there exists many more like PYTHON[12] as some other functional languages too.

[11] The star represents any number of args from 0 to n.

[12] A high-level object oriented scripting language made by Guido van Rossum and Fred L. Drake, Jr. in 1991.

4.9 Topic comparison

4.9.1 Summary table

Before going through the topic analysis, we show in the table 4.1 a comparative summary of the important features of all web languages we saw.

	Kind	**Typing**	**Built-in**	**Portable**	**OO**	**Price(€)**
Perl	Script	Loose	Medium	High	No	0
VBScript	Script	Loose	Med-High	Very Low	No	2850(ht)
C#	Program	Strong	Med-High	Medium	Yes	2850(ht)
PHP	Script	Strong	High	Very High	Yes	0
ColdFusion	Script	Loose	Very High	Very Low	No	1250
JSP/Servlet	Program	Strong	High	High	Yes	0
Tcl/Tk	Script	Loose	High	Med-High	No	0

Table 4.1: Summary of web languages and their notable features.

Some parts of this table need some precisions. The **Built-in** column represents the built-in facilities provided by the language. Also, the **OO** column informs on the possibility to handle object oriented features. Note however that Perl and Tcl have some object oriented features with their *OO* extensions(OO Perl and OO Tcl). And finally, recall that Java is not Open source but its SDK is distributed for free.

4.9.2 Scripting and programming languages

We insisted on the difference between scripts and programs. A program (written in C, Java, etc) is first read and then translated into a machine language and can be run any number of times. A script (written in Perl, PHP, Tcl, etc) will be read and translated each time it is run. In oher words scripts are *interpreted* and programs are *compiled*. Because compiled programs are already converted into machine language, their executions are faster. Furthermore, the source code integrity is guaranteed because the running program is a different file than the source file, but not with scripts where both source and running files are the same.

There exists another important difference between scripts and programs: the debugging tools available. On one hand, for the major part of the scripting languages, debugging is a tricky task because of their loose typing. However, some of them (especially Tcl) counter that with *stack recovery* processes coupled with testing tools. On the other hand, many programming languages, particulary object oriented ones, have powerful step by step debugging tools with environment checking.

However, web-oriented languages like PHP are very easy to use because they have *just what you need*. They are designed for web development and provide many *built-in* facilities also notable. At the opposite, large-scale languages, like Perl, are more complex, they propose a wider range of functionalities and they are more constraining, so in brief, they require much time at the first contact.

4.9.3 Strong typing

Strong data typing could seem necessary every time, independently of the situation. But many scripting languages (the case is more marginal for programming languages) possess a few types of variable and work very well. Let's see the pros and cons of each.

Loose typing permits to save some space in memory (reusing some unused variables cells) and permits also more flexibility and robustness. It is also quick and easy to write. But, the lack of an accurate variable typing makes hard any little change in the code.

Strong typing makes the programmer's life better. Even if such programs are more constraining to write, language specific editors deal with the type checking. Furthermore active code editors check also the syntax in real time letting the programmer concentrate on logical errors only. Lastly, strongly typed languages provide almost always documentation, debugging and integration tools.

4.9.4 Portability

One of the most important feature of a technology is its portability possibilities. From this point of view, each language we see is highly portable, except ColdFusion and ASP. But the .Net technology is forecasting a more *compatible* horizon for Microsoft's fans.

Nevertheless, choosing a Microsoft or a Macromedia technology will restrict the developers on a few platform, but the weight of the portability capabilities is not the same

in everybody's sight.

4.9.5 All-in-one packages

Many technologies are available in all-in-one packages, sometimes for free, sometimes not. These packages are easier to deploy and to configure, decreasing the set-up time of your environment. Almost all technologies we talked about are available in this form.

ASP, .Net and ColdFusion are quite expensive. They continues to seduce developers because of the ActiveX and CFCs objects. They are very attractive and permit many interactive possibilities. ColdFusion has another asset: its WYSIWYG approach that attracts more web designers. Despite its expensive price, more or less one fifth of the web sites are written in ColdFusion. Even if promising free technologies are emerging (especially PHP and JSP-Servlet), the Microsoft's stamp still appear on the biggest part of the world's web servers.

Perl, PHP, Java, Tcl/Tk, here are four different ways of programming with their own syntax and also their inherent specificities. And each of them propose an incredible choice between servers, editors and, at the opposite to costly technologies, between underlying hardware.

4.9.6 Public licences

We will now talk a bit about the new phenomenon of public licenses. A lot of emerging technologies begin to be developed by independent people from all over the world and working in the same way. As it was the case for the incredible birth of Linux, PHP and Tcl to a lower extent, exploded on the web development's universe. And now, many developers add some features, libraries, tricks and experience just to create the best web language in the world. Without entering the public licence debate, we would underline this new movement to create free quality software.

Chapter 5

Common web servers

5.1 A few words of introduction

Every distributed program (as web sites are) needs a *background application* to run. This is called a (web) server and acts as running support for such applications. Servers are used to answer to clients requests (in our case, web browser requests) and manage a lot of them as efficiently as possible. They also are the intermediary between distributed programs and the underlying operating system or hardware.

In this chapter, after a cross-view among the most important interfacing standards, we will see how Microsoft's servers and their rivals work and finally, we will highlight some important assets a server must have.

5.2 Interfacing standards

5.2.1 Common Gateway Interface (CGI)

The *Common Gateway Interface* (CGI) was the first major standard for interfacing external applications with information servers, such as HTTP or web servers. CGI was first developed to permit to display some dynamic content on web pages. But, CGI allows above all to interface servers with an external application. By example, on a search engine page, the server can ask a distant program to search on the web what the client is looking for. CGI defines in fact a *gateway* between HTTP servers and programs

running on them or on other hosts. That enables these servers to create *on-the-fly* content with some client's personal information.

Practically, CGI programs or scripts can be written in C, FORTRAN, Perl, Visual Basic, Shell Scripting and many more. Perl is the most popular language to write CGI scripts. There exist a huge number of reusable CGI apps libraries for many languages (especially for C and Perl) to *do not reinvent the wheel each time*.

In brief, here is the common way a CGI program runs. First, recall that an HTTP request contains some header lines. One of them is used for the address of the web server. In it, CGI programs can find everything they need to create a web page. Let assume that someone asks for a web page with some dynamic part to process. The web server will split up the URL[1] and transfers the request to the right CGI process by the standard input channel(STDIN). In clear, the address (or sometimes the data part) contains some arguments used by the server for sending back the required page and, if necessary, by CGI programs that will create the requested page. After processing its job, the script (or program) answers to the server using the standard output channel (STDOUT) or with environment variables and it can send a whole page or just the dynamic content to the server which finally transmits the page to the client's browser. Note that some CGI environment do not use the standard channels for data input and output but they use ports. However, in both cases, the continuous open-close delays for communication are a pain in the neck.

However, each time someone runs a CGI script, a new instance of the script is created in memory, so if one hundred people ask the same page (needing a script interpretation), one hundred instances of the same script will run. In worst cases like that, the server's memory will be quickly overfilled. Furthermore, this standard involves that CGI applications are out of the server's *boundaries*, so, out of the server's address space. That implies that CGI apps ask some delay to run and some inner information are not *exportable* to them. These execution limits make CGI loosing some fans seduced by other standards like ISAPI-NSAPI or the promising Java standards.

[1]Uniform Resource Locator: unique address for hosts connected to the Internet

5.2.2 Internet Server Application Programming Interface (ISAPI)

The CGI protocol defined a gateway through which a server and some distributed processes can communicate. But, these processes have to discuss through the standard in-out (or dedicated) channels. In order to reduce these lost transmission delays, Microsoft and some external associates decided round ten years ago to develop a standard that puts developers in the core of the server, ISAPI was born. It consists in a large collection of functions that allows developers to extend their servers' functionalities in an almost unlimited ways. Two methods can be used: *Internet Server Application* (ISA) and ISAPI Filters.

ISA are Dynamic-Link Libraries (DLLs) loaded at the server's runtime. They are nothing else than server extensions that can be loaded in or deleted from the runtime memory. They can be called using two entry points: `GetExtensionVersion` and `HttpExtensionProc`. A DLL is loaded when someone requests it for the first time and it will share the same memory space, the same process and the same resources as the server[2]. When a server receives a request, it creates an *Extension Control Block* (ECB) structure defined to hold specific blocks of data and allow some functions to use them. An ECB is nothing more than a C structure used to pass information between the server and its extensions.

ISAPI Filters are closer to the server. They make the server more flexible and nearer the developer's will. A filter is a linked DLL that will be examined each time an HTTP request knock on the server's door. The idea beneath is letting the opportunity to hand a specific case that you are interested in. The filter's job is to sieve the server's notification messages and to give the hand to the filter that has to manage the message. Because filters are dealing with very low-level information, they must have been declared in the server's registry before being executed. They also have two entry point similar to the one used by ISA: `GetFilterVersion` and `HttpFilterProc`.

The two entry points for each technology are working in a similar manner. The `GetVersion` function inform on the API's version compatibility with the server and `HttpProc` are used to load and start the process exactly as a `main` in C.

[2]here is the difference with .EXE type processes.

ISA and ISAPI Filters can be implemented in C or C++, but Microsoft's VBScript, and since lately C#, are the most used as ISAPI in Microsoft environments.

Finally, ISAPI servers provides many built-in facilities and flags to deal with their inner variables, messages and functions. Compared with CGI, the processing time is speeded up because the ISA (or the filter) is present in the server memory and shared processes better support heavy traffic load as well as data sharing.

5.2.3 Netscape Server Application Programming Interface (NSAPI)

Almost at the same time that ISAPI was launched, Netscape, the 90's leader on web applications, answered with its own interfacing standard. As Microsoft's idea, NSAPI permits not only to extend your server but also to modify the server's inner functioning as you wish.

Even if NSAPI strongly looks like ISAPI, the working way is a bit different. First, CGI applications are considered like real API, but are still executed out of the server's limits. All NSAPI functions are built into the server's core and they extend or rewrite its functionalities.

As ISAPI, NSAPI functions are sharing the same process and the same resources than the server, so they allow to deal with low-level server's functions and information. But NSAPI lets the developers free to touch the deep server operating way and configuration, leaving them incredibly more free than in the ISAPI environment. But this freedom requires more programming knowledge and experience.

General principles of NSAPI and ISAPI are the same, but the implementation made by Netscape is a bit different since every NSAPI function can either extend or rewrite some server's functions. An *HTTP Request/Response* process is the whole work starting when someone asks something to the server and finishing when the server answers to the client. This mechanism is divided in six steps[12]:

- Authorization Translation: Any client authorization data converted into user and group for server
- Name Translation: URL translated or modified, if necessary
- Path Checks: Local access tests to ensure document can be safely retrieved

- Object Type Check: Evaluate the MIME[3] type for the given object (document)
- Response to Request: Generate appropriate feedback
- Logging of Transaction: Save information about the transaction to logging files

These steps are gathered in five NSAPI function classes[12]:

- `AuthTrans`: Performs Authorization Translation
- `NameTrans`: Performs Name Translation
- `PathCheck`: Performs Path Checks
- `ObjectType`: Performs Object Type Check
- `Service`: Performs Response to Request and Logging of Transaction

Every function you will write will belong to one of these classes. The server must be aware of which functions take priority on the other and if there exists other additional ones. Even if a request processing is always divided in the six steps explained before and even if their sequencing is fixed, the rank of a specific function into one step is defined in the server's configuration file. So each of the new APIs must be declared in this file to be available for the server.

5.2.4 Java technologies

In 1995, Sun released its first version of Java language and, in its first SDK offers a client-side opportunity to counter CGI apps. The very difference between Java and existing technologies (particulary CGI scripts and VBScript) is that Java is a complete programming language. At that time, scripting languages were in fashion (remember that Perl was the usual way to write CGI scripts even if C or C++ did it nicely) and Java broke the habits. Before Microsoft comes with its ADO, Sun offered the first way to add *real* dynamic content on the web (for example animated objects). Web pages began to embed real applications named Java Applets.

[3]Multipurpose Internet Mail Extensions (RFC 1341) : defines a coding system for mails exchange

Briefly, an applet is a Java program present on a server. On the web page where you want to add an applet, you just need to give the reference of the applet between HTML <applet> tags. When a browser accesses to the page, it will access the server where the applet is and download it. When the browser has finished, it simply displays the whole page executing the applet (using a JVM running on the client side).

Several years later, Sun decided to launch its own server-side language[4]. Because the JSP/Servlet technology is a completely different point of view than CGI and ISAPI/NSAPI standards, the Java Community had to develop its own interfacing standard. As it is Java code, JSP/Servlet have to be processed by a JVM. Also, the server must provide a *Servlet Container* which just act like a pointsman and executes the corresponding servlet[5] according to a requested web address. In that way, the Container takes advantage from the powerful parallel facilities of Java and manages a threads' pool to handel several requests in the same time. This parallel easiness decreased the response time and because everything is managed by the Servlet Container, data concurrency is also controlled by it.

Nowadays, after Sun and Apache, even the Microsoft's server supports JSP/Servlet content.

5.3 Web servers overview

Since many years, the battle for web servers rages between Microsoft and the Apache community. Other companies or Open Source communities tried to enter the fight, but their influence on the market is marginal. We will briefly speak over these two and finally, we will report some notable assets of a server in order to see which one seems more attractive.

5.3.1 Microsoft's Internet Information Server (IIS)

The passed years have seen many releases of *Internet Information Servers*. Each version was given with either NT servers or more recently with Windows operating systems.

[4] see section 4.6

[5] remember that JSP pages are compiled into servlets.

We will just talk about IIS 6 the last version, because it is a complete review of the system kernel which is incomparably more powerful than its older brothers. A new multi-threading system was created to give more stability to the system.

The server is divided in two part: a *listening* and a *processing* part. The first one's job (also composed by threads) is to listen to clients requests and then pass them to a process from the other part. If one process crash, in the worst case, all its part of the server needs to restart.

IIS servers supports a wide range of languages but are working over ISAPI, a standard that no UNIX based server provides. But until now, it is the only one able to handle the .Net technology efficiently, but,unfortunately, IIS only works on Windows environment and is a bit costly.

5.3.2 Apache http Server

Apache servers are the reference in the academic world as IIS are in the commercial one. Some months before the launching of IIS 6, *Apache http server 2* made its first steps. It consisted also in a major rewrite of the server kernel. It uses a *father-child* process. The father has just to control that each request that comes in finds someone to answer it. If a child encounters a problem, a new one is raised. The underneath system is similar to the IIS, but the kernel, so the basic functions set, is really minimal in the Apache case. Their idea is putting the essential to avoid as many failure risks as possible when IIS is integrating more and more built-in functionalities.

Apache is also supported by a large community of developers writing more and more optional modules. It is essentially a NSAPI server but its reputation is to deal with everything (needing sometimes a additional package). Last but not least, in addition to be widely supported, every language interpreter or compiler can be loaded as modules to increase the server's performance.

5.3.3 Remaining firms

Even if Apache and Microsoft are dominating the web servers' world, there exist many other ones available like Sun, TuX from RedHat, Zeus and many more. The major part of them are multi-threaded systems with various performance and various supported

platforms. Recent benchmarks on UNIX-based servers putted Zeus as the most powerful one for pieces of data of maximum 5 MBytes when performance were equal for bigger pieces. But, Zeus is not available on Windows and is a bit expensive.

5.4 Assets of a server

5.4.1 Traffic load tolerance and performance

The first job of a web server is to answer to clients' requests as quickly as possible. A high-performance server has to use its resources as efficiently as possible. It means that average requests must use very few process time and the throughput must be very near to the bandwidth capacity. In that domain, UNIX servers are the best choice as shown in the benchmarks.

5.4.2 Supported languages and modular possibilities

More and more languages and standards are emerging in these Internet's growing days. One of the first assets that people search on a web server now is the range of languages it supports. Also, evolutive capabilities are actively sought to be prepared to a conceivable future major revolution. Even if they are powerful, many UNIX-based servers are limited to CGI apps. IIS and Apache can run web pages written in many languages, but Apache has a unique modular architecture allowing to add many modules of which native languages interpreter and compiler. However, if you intend to go deeper in and test the .Net technology, you have no other choice than choosing IIS.

5.4.3 Supported platforms

Supported platforms range is maybe as important as the choice possibility of the programming technology. Zeus, Sun and Apache are the most varied one. First of them, Apache runs on nearly every UNIX and LINUX platform as well as on Windows. Independent special platform modules are making the intermediate between the underlying operating system and the server's kernel. This architecture reduces the kernel size and optimizes performances. Second, Zeus is available on many UNIX-LINUX platforms and on Mac OS X, but not on Windows. And finally, the Sun's server is highly portable too.

5.4.4 Security and reliability

From a security point of view, UNIX-LINUX-based servers are incomparable. Since the years that such servers are running everywhere in the world, no major virus alerts (or attack in large) were reported. On the other hand, IIS was the prey of all kinds of treats and some attacks paralyzed almost every big information servers in no time[6]. Treats like that are asking more and more attention and, if you do not care about getting a rap over the knuckles of someone when your system bugs, UNIX-LINUX-based and more generally Open Source servers are a better choice.

[6]Recall for example the Sapphire worm in 2003 that infects round 75.000 systems in less than 30 minutes.

Chapter 6

Streaming technologies and protocols

6.1 Streaming and rescuers

Streaming technologies are associated to protocols and their methods for displaying video and audio files on the Internet. Streaming is in fact the process of playing a multimedia file while it is still downloading. Our system offers some data collecting possibilities, but so far, we take only a little advantage of the camera of Aibo. We thought to use Aibo as a *cameraman*. The streaming technologies now give simple ways and methods to send live videos.

6.2 Streaming overview

6.2.1 Streaming types

There are two types of streaming: *progressive* and *real-time*. In the progressive type, the client that starts downloading a video will first wait a while (depending on its bandwidth) before the video begins. Then, while the file is coming to her or his hard drive, the video continues to be played. Note that this kind of streaming does not allow the viewer to see further in the clip because the download is linear. That is why progressive streaming is often referred as HTTP streaming because HTTP servers can perform that.

On the other side, real time streaming answers to the lacks of the progressive mode. Because it is more powerful, it is also more complex. This technology needs a dedicated server and uses special network protocols. Using the real time type, it is possible to send live content such as in live telecast. Furthermore, viewers can jump from one to another part of the file in the case of prerecorded files. This is the typical method for live content.

6.2.2 Delivery methods

Three methods exist to distribute stream media (SM): on demand, live and simulated live.

On demand like video stores, the SM starts when the user clicks on a link.

Live as a TV live broadcast

Simulated live same as live, but the broadcasted event is prerecorded.

For live and simulated live videos, the user cannot fast-forward, backward or pause the clip. For live broadcasting, three ways are available:

Unicasting simplest and most used method. It requires only little or no configuration at all.

Multicasting standardized method for delivering presentations to a huge number of users over a network or the Internet. Much difficult to set up but more efficient.

Splitting the source server (the *MediaServer*) can share its live streams with other servers. Clients connect to these other servers, called *splitters*, rather than to the main *MediaServer*. Splitting reduces the traffic load on the source and enables it to stream other broadcasts simultaneously.

6.3 Early days of stream media

6.3.1 Multicast backbone: the basis of conferences over the Internet (Mbone)

A multicast backbone is "a group of servers throughout the Internet that support the IP multicast protocol (one-to-many) and allow for live audio and video transmission. When it is necessary to leave the Mbone and traverse a part of the Internet that does not support multicast packets, (...) Mbone routers (mrouters) encapsulate the multicast packets into unicast packets and unencapsulate them on the receiving side.[64]"

The Mbone was an experimental network of multicast-capable routers and is used for communication sharing such as conferences or workshops. It is not connected to Internet Service Providers (ISPs) but well to many universities or research institutions. This network will become obsolete as more and more routers supports multicast traffic.

When someone wants to join a conference diffused on the Mbone, she or he connects to a Mbone web site and receives the multicast conference address and the UDP[1] port.

6.3.2 ReSerVation Protocol (RSVP)

This protocol is defined in the RFC 1633 and 2205[26] and is used to request a specific Quality of Service (QoS) from the network for an application data stream. At each node visited by the RSVP request, it attempts to make a resource reservation for the stream, so it is not a routing protocol. RSVP works over the IP architecture.

To make a resource reservation, two local decision modules are called: *admission control* and *policy control*. The first one checks if the contacted node has enough resources available to supply the requested QoS. The second one determines if the user has permission to make the reservation. If either check fails, the RSVP program returns an error notification. Otherwise, the RSVP daemon configures the processes in charge to obtain the desired QoS.

RSVP works with unicast and multicast addresses and maintains *soft* states in routers and hosts to dynamically react to routing or membership changes.

[1] User Datagram Protocol: connection-less unreliable transfer protocol (opposite of TCP)

6.4 Streaming protocols

Streaming protocols uses different mechanisms to work. They roughly need three kinds of protocols to work efficiently:

description defines the structure of the multimedia file or gives connection details to join a SM (sometimes both of them)

control used to supervise the flow of bytes (stop, fast-forward, etc.), the entry of new hosts and the encoding type supported

transfer in charge to send the multimedia file

6.4.1 Session Description Protocol (SDP)

"SDP is intended for describing multimedia sessions for the purposes of session announcement, session invitation, and other forms of multimedia session initiation."[8] SDP was originally designed to work on the Mbone to communicate setup information of conferences. But, it intends to be general purpose to be used also in other environment. It does not deal neither with the content of the SM nor with the negotiation of media encodings.

A SDP message consists in text lines of the form <`key`>=<`value`> including information on:

- the session name and purpose
- the duration of the session
- the media comprising the session
- how to receive the media
- the media type
- the minimum required bandwidth (optional)
- a link to the person in charge of the conference (optional)
- some useful URLs (optional)

- etc.

"Sessions may either be bounded or unbounded in time."[8] In both cases, they may be active at specific times and for time-bounded sessions, SDP can define the list of their activation times.

6.4.2 Session Announcement Protocol (SAP)

To make the world aware of a conference, an announcement is periodically *mulicasted* on an arbitrary well-known multicast address. These announcements can be made by SAP packets embedding SDP messages (or other equivalent protocols). A SAP message can contain only one SDP message.

6.4.3 Synchronized Multimedia Integration Language (SMIL)

SMIL (pronounced "smile")[61] defines a type of documents representing the sequencing and the general organization of a stream media. These documents are XML-based. A SMIL element is identified by an ID number and is structured as the following:

- the `head` part divided in:

  `layout` gives information on the graphical and acoustic arrangement of the files composing the whole media.

  `meta` is a single pair of *key-value* to give a document property (the author(s) for example). The `head` can contain many `meta` attributes.

  `switch` defines a set of elements of which only one will be chosen (for instance same media with different output rates).

- the `body` part contains the list of files and their properties composing the SM.

6.4.4 Session Initiation Protocol (SIP)

SIP messages[20] are used to initiate and handshake connections for multicast conferences or two-party calls. It is widely used on wireless networks. The initiation phase has three goals:

1. locate the terminal where the called party can be reached,

2. agree on a set of media and possible encodings for communication

3. determine if the called host accepts to be reached (or is reachable)

SIP chose an e-mail like identifier for people of the form *user@domain* (the domain can be either a name or an IP address). Using invitation requests, one can asks its SIP module to contact another person or initiate a multiparty conference.

Note at last that SIP is independent of its transport protocol as it provides its own reliable mechanism and (proxy) redirections are available.

6.4.5 Real-Time Streaming Protocol (RTSP)

RTSP[22] starts and controls the flow for stored and live SM. It works on both unicast and multicast destinations. It defines messages that pass through a two-way dedicated channel and drive the progress of the MS (stop, pause, etc.). The figure 6.1 shows the general operation using RTP[2] for transport.

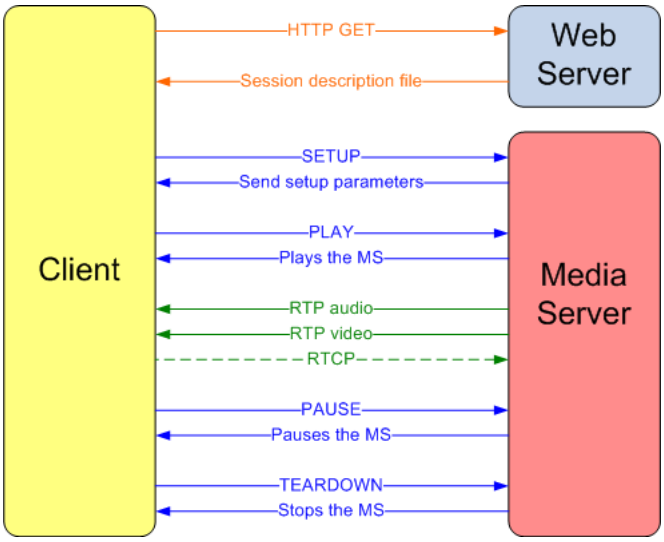

Figure 6.1: RTSP operation overview

[2]see section 6.4.6

The client asks the description file of the media (stored in a SMIL, SDP or any description document). Then the client sets up the connection with the server(s) that share (part of) the multimedia file. Synchronization in case of distributed component does not concern RTSP, it only acts on the progress of the SM.

RTSP is totally independent of the transport protocol used to send the data. Regardless of the description format, each media stream is identified by an URL method, `rtsp://`, that refer to either a single or several media streams. However, using different ports for each stream in case of multi channel media is not supported. RTSP relays on standard HTTP mechanisms for authentication, encryption and content labelling and payment. The server diffusing the media stores session identifier and states so mobile systems pose no problems.

6.4.6 Real-time Transport Protocol (RTP)

RTP is a one-way protocol used for multicast conference or multicast data sending. It provides an *end-to-end* delivery service for real-time data. "Those services include payload type identification, sequence numbering, timestamping and delivery monitoring."[21] Coupling with it, a control protocol (RTCP) monitors the QoS and gives information about the participants in a session. RTP works most of time on UDP but is independent of the underlying transport protocol. However, it relies on it for reliability (if it uses a reliable protocol) or optional resource reservation. But, RTP provides a synchronization mechanisms (with a clock system) in case of distributed mixed data for one SM.

There exists a lot of proprietary and free implementations of RTP. The most known is the Apple one using SDP messages and RTSP control.

6.4.7 Advanced Systems Format (ASF)

Microsoft decided to create its streaming transport protocol that regroups assets of description protocols too. ASF[45] files can contain WMV[3], WMA[4] or a lot of other types of data. ASF contains mechanisms to support efficient playback from media servers

[3]Windows Media Video
[4]Windows Media Audio

and allow authoring control. It is also platform independent (for the receiver) and does not care about the transport protocol used.

An ASF file is divided in three parts (called "objects"):

Header contains general information about the file

Data contains the stream data (maybe divided in more streams dependent or not)

Index (optional) gives *keyframes* and their associated index for fast seeking through the SM

Each object has a globally unique ID and a size value to identify it from the others and fasten object processing on hosts.

6.5 Summary table

The table 6.1 shows a brief summary of the streaming protocols.

Protocol	Type	Independent
SDP	session description	Yes
SAP	advertising	No
SMIL	handshake, file description	Yes
SIP	handshake	Yes
RTSP	control	Yes
RTP	transport	Yes
ASF	description, handshake	No

Table 6.1: Summary comparison of the streaming protocols.

Part II

Our contribution

Chapter 7

Our Contribution: introduction

7.1 Context

7.1.1 Why using Aibo?

The growing *home automation* offers new opportunities for rescue teams. It can provide them information about locations that are difficult to access and then, plan their efforts more efficiently in case of emergency. Their idea is turning to profit the proliferation of technologic devices in modern households, especially all *domestic robots*. That's why Aibo, the Sony's dog combined with other robots, will help a lot, particulary in Japan where it is widely present and the earthquake risk is very large. In this work, we had to view Aibo as an information collector.

As explained in a previous chapter[1], the IRS does research in rescue robots. They build them with a view to helping people in rescue operations following natural disasters. These robots have to be customized to progress in particular areas according to the catastrophe. Therefore, a lot of robots with specific specialities are developed (running, flying, etc.).

Everybody can understand that Aibo is too fragile and weakly composed to help in a corresponding situation, but in robotic research it could be a good start. Results obtained with it in a reduced environment and with small functionalities could be transposed for larger projects. And, above all, Aibo is the most diffused entertainment robot

[1] see section 2.2.2

in Japanese and American houses.

7.1.2 Overview of our job

Our task was to develop a system in which a domestic *Aibo dog* has to collect information according to operator demands, and send them to a *Local Server* (LS). This server has to pack up these data and forward them to a *Web Server* (WS). The WS's job is to show these information on a web page making them viewable for anybody (so not only for rescue team members). Information collected could be pictures, sounds, the ambient temperature or pressure or other piece of data that can give a good representation of the room's condition where the dog is.

The information gathering was not our priority in this project, we had to develop a communication protocol between the different hosts: the dog, a small server and a web server. In sight of being able to show the result of this project, we limit ourselves the dog to take pictures and record sounds around it.

7.1.3 The three hosts

On the Aibo side, we first had to develop a communication OPEN-R object. As its name suggests it, its job is to manage the communication with a (local) server. To this end, we create a protocol to make the communication as safe as possible to send commands to Aibo and also receive files (and other various data) from him. This protocol in place, Aibo can receive a collecting request, process this request and send the result to the server.

Second, we had to implement a small local host that acts as a relay for Aibo. As Aibo only has a wireless connection to communicate, it was necessary to have a server near every Aibo present in the system of "*electronic surveillance*". This LS plays a *middleware role* in the protocol as its job is to forward commands asked by the WS to Aibo and receives results from Aibo according to the request. When a request is sent to Aibo, it receives the files requested and packs them in HTTP POST requests to transmit them on the WS that will at last show these data.

Finally, the *Web Server* has two major goals. The first one is to allow to rescue teams to see the available connected local servers and their Aibos and to send requests to them

(or some of them) through a web page. The second one is to show the requested information on another web page. Until now, anyone can access this page to take attention of the information collected by Aibos, but this can easily be configured[2].

Before going more deeply in the functioning of each host, we will first present the main protocol architecture which bind these three and the general working way of our system.

7.2 A global view of the architecture

Our job was to create a way to *discuss* with Aibo. By discussion, we mean that we had to create a shared *language* between a WS and it. Clearly, we invented a so-called *protocol* to make these two different agents understanding each other.

We developed a reliable way to safely send requests (also called commands) to one or more Aibos, and to recover its or their results. Our protocol, especially its implementation, guarantee a high level of reliability and also a special mechanism to recover information that will possibly fail to pass from Aibo. The figure 7.1 below shows a general view of our system.

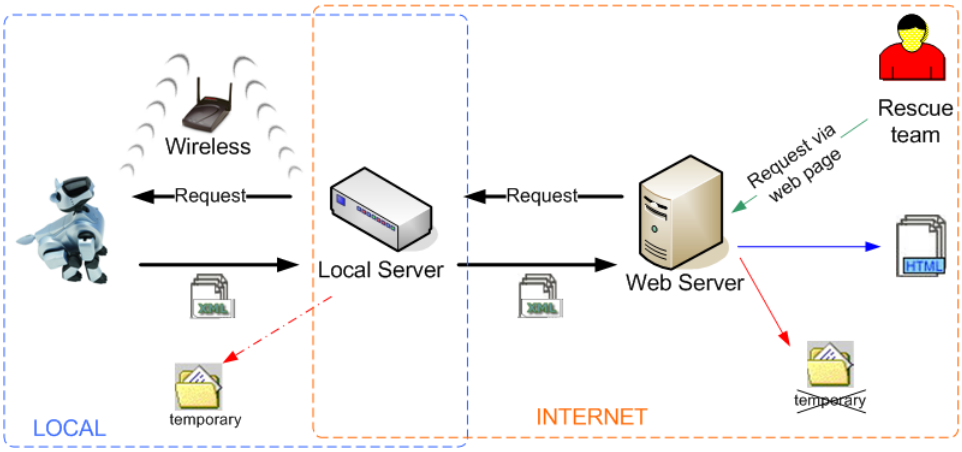

Figure 7.1: Global overview

[2] A login system is already present on the web site, enlarging this system for data display is elementary. See section 10.2.3

Intuitively, when a member of a rescue team wants an Aibo making some information gathering, he connects on the WS. The latter will send the command to Aibo through the LS according to the rescuer demand. The collecting job finished, Aibo contact its LS back, sends its results and the LS is in charge to safely recover and pass them to the WS. If a connection-loss between Aibo and the LS occurs (because of the wireless connection) or, more generally, if Aibo encounters a problem, the server will try to contact it again to obtain the missing files.

On the figure, you see that we use a special XML file for each answer. Let assume for now that this file is employed as header file for every result sent back to the LS and transmitted to the WS[3]. Note also that file saving on the LS is temporary and is only useful to ensure that all result files are effectively transferred to the WS though the Internet.

7.3 Our protocol and its layers

Let's go deeper in the communication protocol created. It is spread across the three hosts: the *Web Server*, the *Local Server* and of course Aibo. Each one is divided in standard ISO layers as shown in figure 7.2.

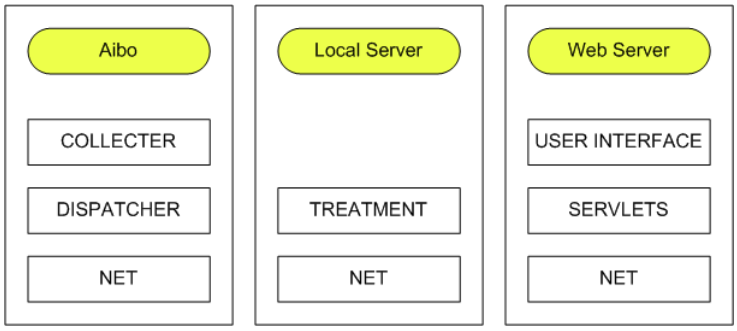

Figure 7.2: Protocol layers

First, the *NET* layer (corresponding to the seven common layers of ISO), the lowest one, is common for every hosts and is in charge of the communication. It relies on

[3]see section 8.2.1 for more details

TCP[4] which provides error-recovery and reliability for network connections. Especially for important data, it is essential to be sure that each of them is completely received by the addressee, so running over TCP was incontestable.

Second, the Aibo side is divided in two more layers: *DISPATCHER* and *COLLECT*. The first of them is the pointsman of Aibo. Its job is to filter commands coming from the outside and pass them to the right OPEN-R object[5]. Therefore, each command known by the *DISPATCHER* will be correctly send to the corresponding object. On top of it, the *COLLECT* layer manages the collecting processes as its name suggests.

Third, the LS has only one other layer above the *NET* one. It roughly has only one job: answer to requests from one or more Aibos and from the WS. No other abstraction layer is needed since each command treatment is on equal footing with the others[6].

Last, the web host is divided in two other layers: *SERVLETS* and *USER INTERFACE*. The first one regroups the whole set of applications that are running in background. These are hidden processes without any *visible* outcome. The last layer, *USER INTERFACE* gathers all apps using the *SERVLETS* layer to create the web site content including for instance the display of the pictures taken by Aibos.

7.4 High-level view of the general operation

A web site must reside on a web server (*software* and *physical*). If the machine where the site resides does not really matter at this moment, the web server influences a lot. Different technologies available were visited in section 5.3 and as explained, the chosen technology will determine the way a connection is managed by the server. As we will justify in section 10.1, we chose the Apache server with the Tomcat extension.

We will see now the typical progress of a collecting request. Figure 7.3 shows the path across the different hosts.

At the server startup, it launches the Servlet Container which will then start a threads' pool[7]. When it is done, the WS is ready to receive requests and goes into

[4] *Transmission Control Protocol* defined in the RFC 793

[5] see section 3.2

[6] Note that a priority policy is conceivable but is not related to abstraction layers.

[7] This a set of Java `Thread` objects used to answer to client solicitations. See section 5.2.4

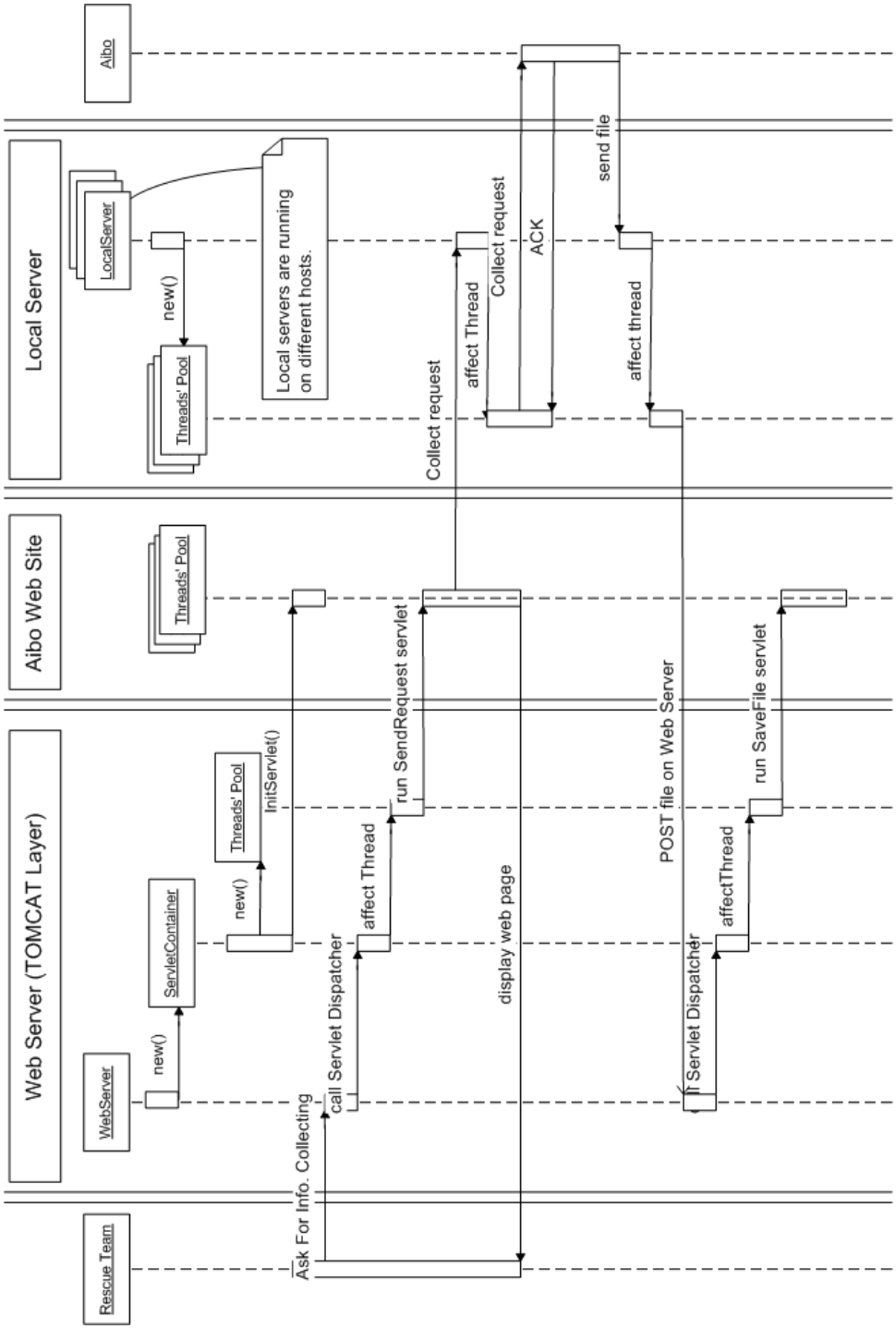

Figure 7.3: Sequence chart of a typical collecting request

sleeping mode. When a member of a rescue team connects to the web page and asks to an Aibo to collect some data, for instance a picture, the request comes first at the server's gate. By decomposing the asked URL, the **Web Server** object remarks that it has to contact a servlet and thus call the dispatcher in the **Servlet Container**. Afterwards, the dispatcher orientates the request to the servlet in charge of collecting requests and affects a thread to the servlet (if possible). Note that when every thread is busy, the **Servlet Container** keeps a *first-in-first-out* queue. The servlet first contact the *Local Server* where the wanted Aibo is registered and then acknowledges of the success or not of the operation to the rescuer.

If the LS is connected, it has instantiated a set of threads to deal with incoming calls more efficiently. When it is called for a gathering command to Aibo, the **Local Server** object (this is in fact a gate which dispatches commands at the entrance of the LS) forwards it to Aibo and ensure that Aibo receives it. Once, the piece of data is collected, Aibo will send back its result(s) to its LS. With HTTP POST request, the LS forwards then all files to the WS. As for a rescuer connection, the **Web Server** will contact the **Servlet Container** and then, the right servlet (**SaveFile**) will do its job.

Every collected file from Aibos present on the web site are visible on another web page. A rescuer or anyone else can connect to the WS and, in a similar way, the right process will display what she or he was searching for. For other type of demand, the process is always very analogous.

7.5 Protocol messages

Let us now explain how the three entities communicate. We added to the figure 7.2 the types of messages that they three can exchange. The figure 7.4 shows these messages with their direction.

1. Aibo $\rightleftarrows$ Local Server (LS)[8]

   $\rightarrow$ **AIBO_READY** When Aibo is running and available for doing some work, it sends this message (when it finished processing a request for example).

[8] Arrows show the direction of the message.

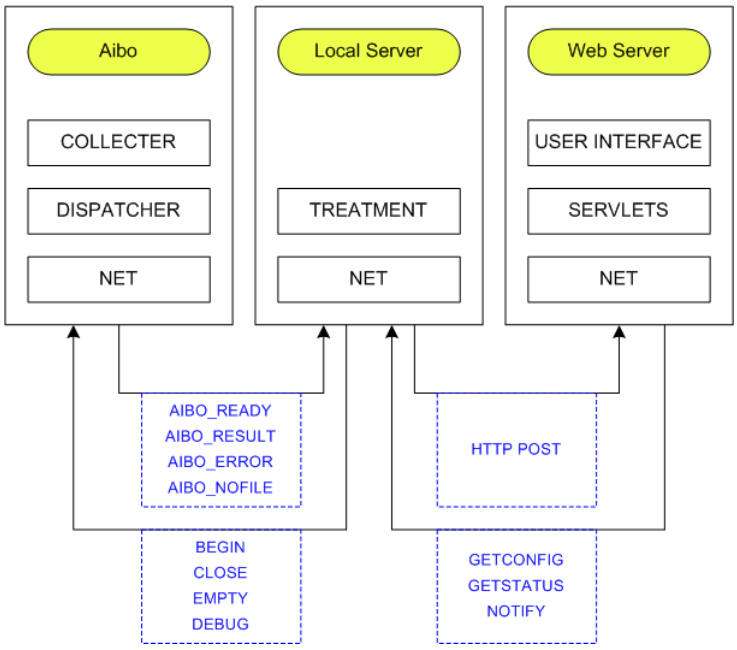

Figure 7.4: Protocol layers with communication messages

→ **AIBO_RESULT** When Aibo has finished its collect operation, it first sends this message to warn that result files will come and also how many files will be transferred.

→ **AIBO_ERROR** This is the message sent when Aibo encounters a problem during a job.

→ **AIBO_NOFILE** It is used in case of a problem recovery if Aibo has no files that were not successfully sent.

← **BEGIN** This message asks Aibo to collect data.

← **CLOSE** When the LS is sure that all files have been sent, he asks Aibo to close the current TCP connection.

← **EMPTY** This message orders to Aibo to empty its collect directory. It is typically used before a new collect request.

← **DEBUG** If Aibo encountered a problem (connection-loss event for example), the LS asks it to send all messages that were not transmitted.

2. Local Server (LS) ⇄ Web Server (WS)

   → **POST** Whenever any file (text, XML, JPG, etc...) must be sent to the WS, the LS uses HTTP POST requests and add its files in the data part of the HTTP packet(s).

   ← **GETCONFIG** Every LS has a special XML file where all configuration parameters are stored (IP address, registered Aibo(s), etc.). Sometimes, the WS need to update its version of the configuration file and asks the LS to give it.

   ← **GETSTATUS** As it will be explained in section 8.3.2, Aibo can be either ready to work or switched off. The LS is always aware of the status(es) of its Aibo(s) and every defined lap of time, the WS asks them to display only running Aibo(s) on the web site.

   ← **NOTIFY** When a rescuer want one or more Aibos to gather some data, the WS will send this message to start the information collecting by this or these Aibo(s).

We pictured here the general functioning of our application and we explained our protocol developed and its way of working. The next part of the work pretends to describe precisely and in a logical order each one of the three *agents* depicted briefly here, one chapter devoted for each one of them. As this protocol can be used in a complete system of electronic surveillance, we present in a last chapter some possibilities of improvements that can be added to upgrade the protocol and its application.

Chapter 8

Local Server

As described in the previous section, the *Local Server* is a middleware server in the communication protocol. This host is registered to the *Web Server* and can handle a set of Aibos. The Local Server can be seen like a data manager which exchanges data (requests or files) between the WS and a specified Aibo according to the header of the request.

This chapter is divided into four sections. The first one presents our choice of technologies, the second one the philosophy of our LS. The third part explains the principle of message passing and the general treatment of data in the server. Finally, we describe the encountered problems and our solutions.

8.1 Strategic choices

8.1.1 Technologies choices

The importance of the LS task implies a light and robust technology which can run in a Linux or Windows environment that can quickly manage a network. The server has to be portable. For these reasons and because of our good knowledge in this technology, we have chosen to develop the LS in Java. This language offers a large panel of network functionality and its byte codes are executable on multiple-platforms according to the JVM installed.

According to the project requirements, the *Local Server* has to be a multi-platform

server. It can be in different places using different operating system. As we have chosen to develop it with Java technology, there is no problem to run it on Linux, Windows or MacOS X. Since the LS has to handle file transfer, it is recommended to use Linux or MacOS X for their powerful capacity of network devices.

Our development domain forces the integrity and the stability of network communication. Thus we chose over the TCP protocol. The only way to communicate with Aibo is the wireless IEEE 802.11b technology. As the LS aims to be a stand-alone tool of communication, it can open simultaneously two TCP connections, one for a wired support through an access point and another one for a wireless support with AD HOC mode. So, it is not necessary to be able to reach an Access Point for starting the server, this one can directly use a wireless card.

8.2 Mechanisms, structures and configuration

Some decisions have been taken to ensure stability and reliability of the *Local Server*. As said, this host is registered to the *Web Server* and registers a set of Aibos. It was necessary to define a structure to memorize this registering information. Each registered Aibo needs to be handled with precaution, for this reason a status mechanism has been created.

These decisions have been taken to ensure an external interface. We had also to develop an internal interface to ensure to true treatment of a client's request. A message passing mechanism has been developed in this way.

8.2.1 Internal interface

The figure 8.1 shows the DTD of the configuration file:

The server's configurations are saved in a XML file that could be parameterized each time the server is launched thanks to a graphical user's interface. This XML file contains technical information like IP, PORT for wired and wireless support, IP of the *Web Server* where the the LS is registered, servlets address on the WS, directory's path for received files and the list of registered Aibos.

The server can manage several Aibos in the same time. Each one is identified by

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT config (localip, localport, wifiip, wifiport, address, webaddress, aiboservlet, xmlpath, aibopath, ifwifidata, aiboinf+)>
<!ELEMENT localip (#PCDATA)>                <!-- IP of the Local Server, wired support -->
<!ELEMENT localport (#PCDATA)>              <!-- Port of the Local Server with wired support -->
<!ELEMENT wifiip (#PCDATA)>                 <!-- IP of the Local Server with wireless support -->
<!ELEMENT wifiport (#PCDATA)>               <!-- Port of the Local Server with wired support -->
<!ELEMENT address (#PCDATA)>                <!-- Location of the Local Server, String -->
<!ELEMENT webaddress (#PCDATA)>             <!-- IP of the Web Server -->
<!ELEMENT aiboservlet (#PCDATA)>            <!-- Path of the java servlet on the Web Server -->
<!ELEMENT xmlpath (#PCDATA)>                <!-- Directory's path including xml files -->
<!ELEMENT aibopath (#PCDATA)>               <!-- Directory's path to save aibo results -->
<!ELEMENT ifwifidata (true|false)>          <!-- Boolean to denote if Local Server support can be wireless -->
<!ELEMENT aiboinf EMPTY>
<!ATTLIST aiboinf  name   CDATA  #REQUIRED  <!-- Name of Aibo, String -->
           ip     CDATA  #REQUIRED          <!-- IP of aibo -->
           port   CDATA  #REQUIRED          <!-- Port of Aibo-->
           spec   CDATA  #IMPLIED           <!-- Specility of aibo, by final integer (PICTURE or SOUND) -->
           address CDATA  #REQUIRED>        <!-- Location of aibo, String -->
<!-- IP is 4 bytes in decimal separated by dots -->
<!-- Port is a decimal number between 2000 and 65536 -->
```

Figure 8.1: DTD of the configuration file

a name and has to be registered on the LS. The Aibos information (name, ip address, port, location, specialty and status) are saved in the configuration XML file, see table 8.1.

We choose XML technology because it is fast and efficient for parsing and transmission. By the way, the XML file is frequently posted to the *Web Server* which can know several *Local Servers*.

At start, the Server parses the XML file to recover Aibos information and saves it in an **AiboData** structure. After parsing, those structures are queued in a Vector which will be progressively used and updated according to the events incurred by the server. The structure is described in table 8.1.

This information is often used by the Thread Factory to process activity with Aibo (see section: Thread Factory). The last field (*BugID*) is used to recover the context request ID (a request ID is a technical prefix to name files received by Aibo, it represents the date with the following format: *YYYYMMDDHHMM*) when a connection loss occurs.

The main Aibos data for the LS is its status which can be OFF (*Aibo is unreachable*, 0), READY (*Aibo is ready to receive a request*, 1) or WORKING (*Aibo is handling a request*, 2). In this project environment, Aibo's status is an important parameter which

needs to reflect with exactitude and in real time (as much as possible) the correct value because it determines if a request can be forwarded or cancelled. Thus it was necessary to define a process to guarantee the updating of this status along the time. We created three mechanisms.

First, at launch, each registered Aibo is called to test its availability. According result of this request, Aibo's status is initialized and becomes OFF or READY.

By protocol definition, an Aibo can be notified of only one request at the same time. For this reason, AIBO's status in the *Local Server* becomes WORKING after a request. This monopolization functionality is defined "Synchronized" to exclude concurrency of request. This WORKING status works like a lock. After the request treatment, status becomes READY again. This procedure constitutes the second way to update the AIBO's status.

Sometimes, an Aibo is not requested for a time. In this case its status is never changed even if it becomes unreachable. For this reason, we have developed a Countdown thread to update periodically the AIBO's status. Each 10 minutes, it checks if all registered Aibos, which are not WORKING, are reachable or not. If an Aibo is unreachable, his status becomes OFF until the communication becomes possible again.

8.2.2 External interface

The Waiting Queue

The *Local Server* manages the client's requests using a robust architecture (see table 8.2). The client is the *Web Server* or the LS itself. When the latter receives a client's request via the wired or the wireless interface, it creates a **Message** and puts it into a queue. This queue stocks each message transmitted by the network interface waiting for a thread to handle it. A thread factory ensures the message handling. Each message could ask a treatment to the WS or to an Aibo.

The *Waiting Queue* (WQ) stocks some messages created by the network interface from a received request or by a thread of the thread factory. A **Message** is defined with three fields as shown in the table 8.2.

More precisely, the WQ is a vector (Java Vector) of **Message**. We have decided to use

Field name	Description	Type of data
Name	Aibo's name	String
IP	Aibo's IP	String
Port	Aibo's port	int
Speciality	Aibo's speciality	final int
Place	Aibo's location	String
Status	Current Aibo's status	final int
BugID	Optionally, contains a request ID	int

Table 8.1: AiboData structure

Field's name	Description	Type of data
Request	Type of the client's request	String[a]
Args	Arguments attached to the request	String[b]
Socket	Java Socket opened with the client	Socket

[a] **FINALIZE, GETCONFIG, GETSTATUS, POST** or **NOTIFY** as described in section 8.3.2

[b] see section 8.3.2

Table 8.2: Message structure

a Vector because this type proposes a practical and structured handling of component, for all type. This vector is shared between the different server's components. Because of that, the handling of the WQ is quite delicate. A message is put by the network interface when it receives a client's request or when it has to send a received file from a registered Aibo to the *Web Server*.

To manage the WQ, we decided to use a well-known design principle: the producer/consumer[1] view, but with a practical particularity. In our system, the producer and the consumer could be the same agent. So, a message with a HTTP POST type is only created by a thread from the server, but not by the network interface.

So, the WQ object is an implementation of the producer/consumer mechanism. Two actions are necessary to respect the theory, one to add the produced Message and another one to consume these messages. These both methods are declared below :

- `public synchronized void PutMessage(Message msg)`
- `public synchronized Message GetMessage()`

`PutMessage()` puts the Message, passed in argument, in the WQ and notify all threads using the Java method: `NotifyAll()`. The consumers are threads from the Thread Factory. While they are living, they try to get message from the WQ with the synchronized method: `GetMessage()`. This method removes and takes the first message from WQ. If the queue is empty, the thread sleeps until it is notified by a producer.

Both methods, `putMessage()` and `getMessage()`, are defined synchronized to avoid the concurrent access to the WQ.

The Thread Factory (TF) is a pool[2] of five threads which are created and started with the Server. The threads' role is to process the client's requests contained in a Message from the *Waiting Queue*. The threads try to get a **Message** in the WQ, each request received by the *Local Server* is handled by one of these five threads. In that way,

[1] *The producer/consumer mechanism: the producer is an agent which creates a data (here a Message) and shares this data. The consumer is another agent which takes and consumes these data's in the shared memory. The producer can add data while the memory is not full, on the other hand, the consumer can consume a data only if a data exists in the shared memory. This rule has to be respected when producer/consumer mechanism is implemented.*

[2] a Java Vector is chosen again for its easiness and robustness of usage

the *Local Server* is very easy to program, it has only to receive the clients' requests and to add them to the queue.

When a thread gets a message, it analyzes the type field and processes the request according to this type. The types that can be processed by a thread are the four MFART messages explained previously (**POST, NOTIFY, GETCONFIG, GETSTATUS**) and an inner technical one: **FINALIZE**.

We can divide the type in two categories: communication with the WS (**GETCONFIG, GETSTATUS and POST**) and communication with Aibo (**NOTIFY**). **FINALIZE** is an internal LS operation which gives the order to die to the thread.

The choice of thread's quantity is minimalist. In fact, this number can be changed according the quantity of registered Aibo on the server. For a small set of Aibos, five thread are sufficient to ensure a quick and punctual treatment of the Waiting Queue's messages.

8.3 System explanation

This section explains the treatment of a client's request by the *Local Server*. All internal mechanisms are currently known, so the general treatment of a request can be presented. Two things need to be clarified, the message propagation in the LS and the thread handling.

8.3.1 Message propagation

As explained before, when a client request is accepted by the server, this one creates a Message. This Message is constructed with the received request divided into two parts with the following structure: `reqType args ref`:

- `reqType`: the type of request can be **FINALIZE, GETCONFIG, GETSTATUS, POST** or **NOTIFY**.
- `args`: set of arguments necessary for the thread handling see 8.3.2. These arguments are a String composed of tokens separated with semicolons.
- `ref`: reference of the opened client's socket.

The network interface puts the Message in the *Waiting Queue* and all threads are notified. The first available thread gets the Message and handles the request according its type.

8.3.2 Thread's handling

Each thread can handle three types of requests, according to the originating agent. It could be a internal communication which concerns the thread itself, a communication with the *Web Server* to update the server or Aibo's configuration or it could be a communication with an Aibo. In this case, it is a request to an Aibo to ask information.

Internal communication

This type of request records only one type: **FINALIZE**. It is just a proper way to kill the thread when the Server is stopped. For this reason, there are no arguments with this type of message.

When a thread receives this type of message, it changes its *live* boolean value to false. Using the FINALIZE type is the only way to change the live's value. When the server is stopped, for sample to change configuration file, messages have to be processed before the system is shut down, which is ensured by queuing a FINALIZE request. With this, we avoid that and also some "lock" on resources problems.

Communication with the *Web Server*

In this case, we can find three request types; two for updating and one for results: **GETCONFIG, GETSTATUS** and **POST**.

GETCONFIG: When this type of message is received, the thread used the DataOnNetwork object (which manages the file sending and receiving) to send the server's configuration file using the opened Java socket contained in the message.

GETSTATUS: As explained in the WS description10.2.4, visitors of the web site can ask Aibo to make some collecting operations. The only constraint is that Aibo has to be ready to be able to process the request. In that way, the web page displays only Aibos known to be connected (actualized every ten minutes, as said before).

Thus, in case of a **GETSTATUS** request, the thread sends a String with each Aibo's status separated by a white space using the opened Java socket contained in the message too. Both types of request ensure a good coordination between the three hosts of the protocol: Aibo, the *Local Server* and the *Web Server*. They do not required any arguments

POST: This type of message is produced by a thread itself, it orders to send a received collected data from an Aibo to the WS. A **POST** message requires arguments that indicates which type of data have to be sent to the *Web Server*. These arguments can be structured following two ways:

1. The data is a file (PICTURE or SOUND): **file**;path;fileName
   - path: directory including the file
   - fileName: name of the file.
2. The data is a rough text: **text**;some text

As the WS runs over Tomcat, following the HTTP protocol, this **POST** request uses the HTTP POST messages to send the file or the data. So, an HTTP header is sent, followed by the file name, in case of file sending, and finally the data is sent (of *DataOnNetwork* type).

8.3.3 Request handling

The most interesting type for the protocol is **NOTIFY**. This type creates the communication with Aibo and permits to send a collecting request and to receive its result.

This type of message is used to forward some requests from a user of the Web Site to an Aibo. The arguments of this type of message are structured like this: **idAibo;requestAibo;requestID;command;arguments**

- idAibo: identify number[3] of the requested Aibo.
- requestAibo : technical request for Aibo[4]

[3]This argument is a String which is cast to an Integer

[4]At this state of our protocol, the request can only be BEGIN. This is the request to ask Aibo to collect information as explain is section 9.2.1

- requestID : identify of the current request[5]

- command and arguments : collect request for Aibo[6]

The processing of the Aibo request and the reception of its result is systematic, all required operations are included in the AiboProcess Object. So, when a thread has to handle a **NOTIFY** request, it has to create a new AiboProcess object specifying the desired Aibo, the server's reference and the server context.

Before starting to communicate with Aibo, the thread has to test if requested Aibo is not WORKING. If it is case, the thread creates a new Message, exactly the same than which is currently process, and adds it to the Waiting Queue. So, the current collect request will be send later to Aibo when it is READY again. Otherwise, the thread has to locks the Aibo to prevent other request to reach this Aibo. The monopolizing method is declared synchronized as explained before.

After Aibo is monopolized, the thread tries to connect to Aibo with an AiboProcess method. If a connection loss occurs during the processing, the request ID is saved in the **AiboData** structure, as explained previously, in purpose to recover the missing files.

So, the thread sends the collecting request that begins using the command BEGIN. The thread is now waiting for data from the Aibo. The first data from Aibo is the number of file that it will send. With this quantity, the thread can begin to receive all collected files using the *DataOnNetwork* object.

The received files are named with the request ID field contained in the **Message** structure. After saving them, the thread creates a **POST** message and puts it in the waiting queue. Another thread will process the sending operation to the *Web Server*. Each file is sent to the WS via another thread to maximize parallelism and reduce delay. When this process is finished, the connection with Aibo is closed and this one is available again for a new request.

[5]It represents the current date with the following format: YYYYMMDDHHMM

[6]Both arguments do not have any signification for the *Local Server* which transmit it transparently to Aibo

8.4 Encountered problems

During our development, we have encountered two main problems. The first one is directly bound with our application domain. Often in rescue system, could a connection loss occur; wireless could encounter difficulty across wall or interference. This possibility had to be handled to prevent the loss of data. A second one concerns the server itself. When the server's manager has to change some configuration, it is necessary to restart the server. This closure has caused some context problems which have been solved as explained in this section.

8.4.1 Connection lost management

A rescue system environment requires a management of connection loss. In a damaged area, it is possible that this type of problem occurs. To promote more stability, we developed a recovery system which permits to recover some loss files. This connection lost problem could occur if an Aibo or the server crashes during a request managing or because of interference in the wireless signal.

This recovery was made in two steps. Each time a request is sent to Aibo, the request's ID is saved in the last field of the **AiboData** structure. So, when the connection is found again, the server knows what was the request's ID and can save the recovered files correctly.

Then, before each collecting request on Aibo, a DEBUG request is sent to prevent a possible bug of Aibo or a connection loss in a previous request. Aibo knows which files had been already sent (see explanation in next chapter), so, when the thread sends a DEBUG request, Aibo replies with the number of missing files. According to this number, the server begins to receive the files. This method ensures to recover all collected files produced by Aibo.

This mechanism was created at the end of our development, so it is not complete. At this moment, an Aibo's crash can be solved but not a server's crash, because the request's ID is saved in a variable which lives only during execution time. A best solution will be explained in the improvement section.

8.4.2 Server's context

To help the server's manager, we have developed a human interface. This interface displays some running information like the countdown thread, the registered Aibos and their states and all operation handled by the five threads and the server itself.

This task was possible because each thread keep a reference of the server and the human interface. So each thread can display its process using this reference. But, when the server is closed to change some parameters, the threads are still running in the system and continue to display some information. To force a thread to stop his activity, we have created the FINALIZE request. When the thread handles this type of request, it finishes its life.

But, while the thread has not received this type of request, it continues to display messages on the human interface, because it has still the good reference.

This difficulty was solved in adding a nonce that identifies the run of the server. When the *Local Server* is launched (class *ServerFactory*), this nonce is incremented and transmitted to each thread from de pool. The thread can display information on the human interface only if its server's number of context correspond with the current number for the server.

Chapter 9

Development on Aibo

This section presents the different mechanisms developed an Aibo to finalize the communication's protocol. Aibo aims to be a collect agent information in its environment. It's the final objective of this project. Our part was to create the communication's protocol, so the information collected is at this point minimalist. As we wanted to present a complete development, we have simply implemented pictures and sounds. Data collected are displayed on the Web Site almost in real time using our protocol.

As explained before, OPEN-R is a set of independent objects communicating with a message passing mechanism. Our development is a set of four OPEN-R objects: *NetCom, Collect, ImageObserver* and *SoundRec*. *NetCom* manages the network communication using the IPStack presented in section 3.2.4. *Collect* dispatches the collect requests and collects the results. *ImageObserver* intercepts a picture from the camera and saves it in a BMP file. *SoundRec* records sounds and saves it in a WAV file.

Before explaining these objects, a structured formalism to describe an OPEN-R object is presented.

9.1 A finite state automatons formalism

All examples provided by Sony use a formalism close to finite state automata to describe the working cycle of an object. Objects have different states and at least the starting state IDLE which is a no-operation state is essential for the system coherence.

As a general design rule, an object cannot be in two states at the same time. The automaton changes from a state to another by a transition. A transition is activated by a message and can have multiple paths leading to the others states in a tree-like way. Each branch of the tree has a condition. The conditions of the branches coming from the same node must be exclusive (in other words, the automaton is deterministic). The automation gets over transition when this transition has been activated by the proper message and the conditions of the path leading to the target state have been satisfied. There is a particular case : the automaton always goes through a transition which has a single path and a condition equal to one.

The ENSTA[24], a French school which makes research in robotics and with Aibo uses a graphic representation of the finite state automata formalism from Sony. We have decided to use it to present our OPEN-R object because it gives a good view of the object operation. In this representation, the different states of the object are represented by circled text. As a convention, the IDLE state is always the beginning state.

Messages are represented by dotted arrows. They come in and get out the object through gates.The arrow-like shapes representing these gates indicate if they are incoming or outgoing gates. Next to the gates, three labels provide additional information : the first one (bold face) is the name of the gate (which is usually the name of the observer) and the second (bold faced and italic shape) is the type of the message exchanged. When it exists the third line is the name of a function. In the case of an incoming gate this function will be called each time a message is received, in the case of an outgoing gate the function will be called each time an Assert Ready (AR) is received.

This is a basic event-driven kind of programming, which is a classical way of handling action-reaction binding in an object-oriented framework.

The transition between two states, which is represented by a full line arrow with a black square in the middle, has two requirements:

- The transition must be activated by the indicated event (receiving a message or an AR).
- The condition on top of the black square must be satisfied.

Going through the transition can possibly trigger one or two of the following actions

:

- Process the code which is under the black square

- Send an outgoing message (represented by a dotted arrow starting from a black dot located on the transition arrow, after the black box)

We have added a type of notification to simplify some complex representation of object. So, to represent the message outgoing to the IPStack Object, a box labelled with IP is drown.

All these specification of the graphic formalism are summed up the figure 9.1.

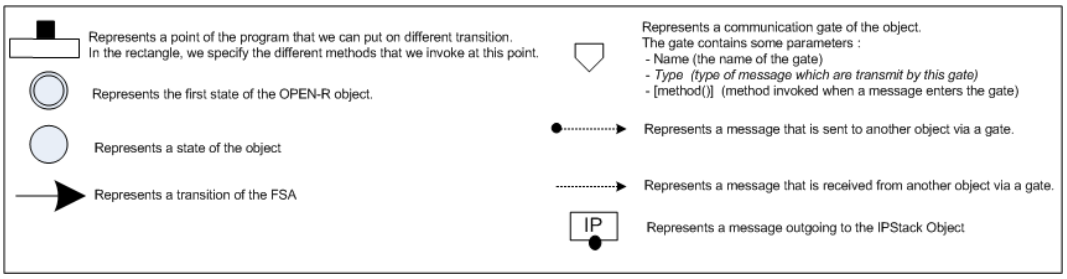

Figure 9.1: Legend of the finite state automatons formalism.

We think that this formalism is perfectly adapted to represent OPEN-R objects. But it is not the only one available, this formalism is comparable with the formalism SDL and UML State Diagram. This choice explained, we can use it to described each objects we have developed.

9.2 Description of our OPEN-R objects

Each objects presented on figure 9.2 will be explained in this section, excepted *OvirtualRobotComm*, *OvirtualRobotAudioComm*, *IP* and *Power Monitor*. These objects are OPEN-R system objects and have been explained in the chapter which presents OPEN-R technology.

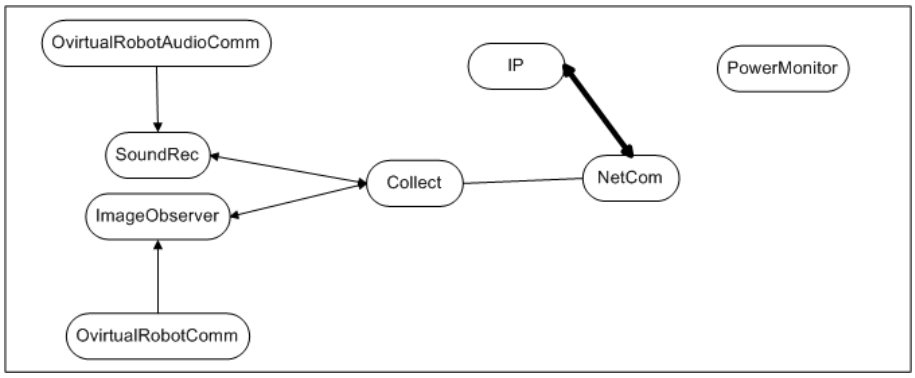

Figure 9.2: Our architecture on Aibo.

9.2.1 NetCom object

This is the main object of MFART protocol. It is a low-level object handling network communication over the IPStack protocol. The *NetCom* object ensures reception of request from LS, dispatching of these requests, communication with the Collect object which produces collected files, sending of these files, and management of connection losses.

But this object does not know anything about the collecting operation and about the results produced. It is only a middle agent which receives a request, transmits it, receives results and sends then back to the *Local Server*. That's why we called it a "low-level object".

As shown on figure 9.3, *NetCom* object could be in one of five states: *IDLE, START, WORKING, SENDING and SENDING_FILE.*

Request managing

IDLE state is a launch state which starts the Server of Aibo. Directly after boot or when a connection with the *LS* is closed, the Listen() method is called and makes the *NetCom* object listening on a predefined port. The state becomes quickly *START* to wait for a client's request. When a connection is accepted, the method *Receive()* is called to receive the client's request. The *NetCom* object knows four types of request: *close, debug, empty, and begin.* The dispatching of these requests is handled by the method *ManageRequest().* According to the request four transitions can be traversed.

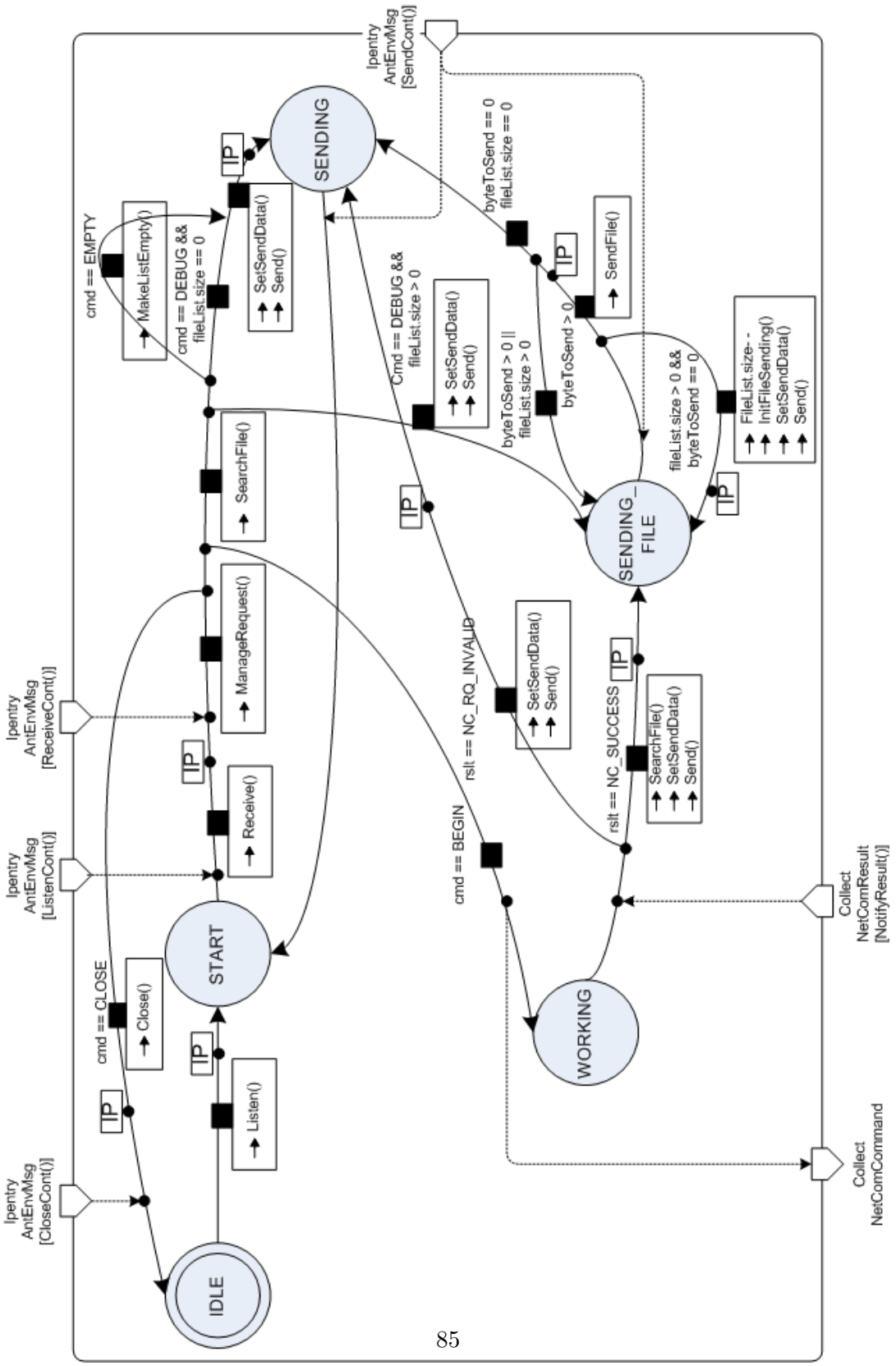

Figure 9.3: NetCom OPEN-R object

NetCom send a *Close() message* to the IPStack protocol to close the connection. It is often used to inform that processing with LS is finished. When this operation is processed, the state comes back to *IDLE* with the possibility to restart the Server.

When Aibo collects information (see section 9.4), all files are saved in a predefined directory. After, *NetCom* sends all files from this predefined directory (it is not the case every time, because of debugging request which is explained just after, but for comprehension, we can consider that all files from this directory are sent). So, to be able to manage other requests, it is necessary to make this directory empty to save other collected files and to avoid overloading Aibo small memory. The **EMPTY** request handles this cleaning operation and lets the *Local Server* know about success of this operation. *NetCom* becomes *SENDING* after this cleaning operation and after it has sent an answer, it becomes *START* again.

The **BEGIN** request asks to Collect object to process an information collect. *NetCom* becomes *WORKING*. It is important to understand that it does not know about the request content, it is the concern of *Collect*. So, the *NetCom* object becomes waiting for an answer from the Collect Object. This answer could be a *NC_SUCCESS* message to inform about the success of the operation; in this case it becomes *SENDING_FILE* after having filled a vector with the references of the file to send. Otherwise, the answer is a *NC_RQ_INVALID* message and *NetCom* informs the LS of the failure.

File Sending

Before entering *SENDING_FILE*, *NetCom* sends a **AIBO_RESULT n** message to the LS using method `Send()` where "n" is the number of files that will be sent. When the LS acknowledges the reception of "n", *NetCom* begins to send the files one by one preceded by a header including the file name and size.

The "n" files references are contained in a vector named *fileList*. According to OPEN-R network management, the files are sent by packet to the IPStack OPEN-R object which handles the transmission to *Local Server*. We have defined the packet size to 512 bytes. The *byteToSend* variable hold the number of bytes remaining. It permits to know if a file is completely sent or not. The files sending mechanism obeys following rules :

If the *fileList* size is greater than zero and the *byteToSend* equals zero, it means that

it remains files in the vector to send. So, the *fileList* size is decremented and the sending file is initiated : `SetSendData()` will set *byteToSend* to the size of the file.

If *byteToSend* is greater than zero it will say that a file is sending. So, `sendFile()` method is called to send the file packet by packet. After this operation, the *byteToSend* variable is decremented and tested. If it is greater than zero or fileList is greater than zero it means than the file is not completely sent or all the files are not sent. In this case, *NetCom* becomes **SENDING_FILE** again to continue the sending operation. Otherwise, it means that all the files are completely sent, and *NetCom* becomes *SENDING* again. When the IPStack protocol asserts the sending, *NetCom* becomes *START*.

Connection loss

As already explained in the *Local Server* section, it is possible that trouble occurs while files are being sent. Since we are designing for a rescue application, our protocol has to recover from connection losses.

When Aibo collects pictures or sounds, it saves files using a technical prefix name: **DEBUG** (for example, if the file saved is called _01.BMP, physically, it is saved as DEBUG_01.BMP). After the file is sent to the LS, *NetCom* renames the file removing the technical prefix (for example, DEBUG_01.BMP becomes _01.BMP). In that way, if a problem occurs during the sending operation, all files that are not fully sent, are always called with the technical prefix.

The role of the *debug message* is to send these files to the *LS* with aim to recover all collected files. So, the function `SearchFile()` is called to save the non-sent file name in the vector. If the vector's size equals zero, *NetCom* only sends a message to the LS to inform that all files were already sent before. Otherwise, *NetCom* becomes **SENDING_FILE** and begins to send files as explained in the *NetCom* description.

9.2.2 Collect Object

The main idea of our project was to develop a protocol of communication between Aibo and the WS. Above this objective, the protocol will be used to make Aibo collecting information in its environment. The collect object was constructed in this way, it is the manager for the collect operation. All **BEGIN** messages received by *NetCom* are

transmitted to be processed by this object.

Mechanism

Collect object is described by the finite state automaton on figure 9.4. This object has two states, *IDLE* and *PROCESS*. When the system is launched, *Collect* is *IDLE* waits for a message from *NetCom* object.

Each time a message occurs, if the object is not processing, the message's type is analyzed, otherwise a message is sent to *NetCom* to warn the disability. According to the message type, a corresponding function is called to start collecting and Collect object becomes *PROCESS*.

When the collecting is done, system calls `Notify()` function which catches the results and sends it via a message to *NetCom*.

In our current implementation, Aibo can only take pictures and record sound. The pictures or sounds collected are saved in the *collect_dir* as planned by the protocol. It is sufficient to add some *type of information collecting* in the collect object, implement a method for this type and construct the specified object to collect another type of information.

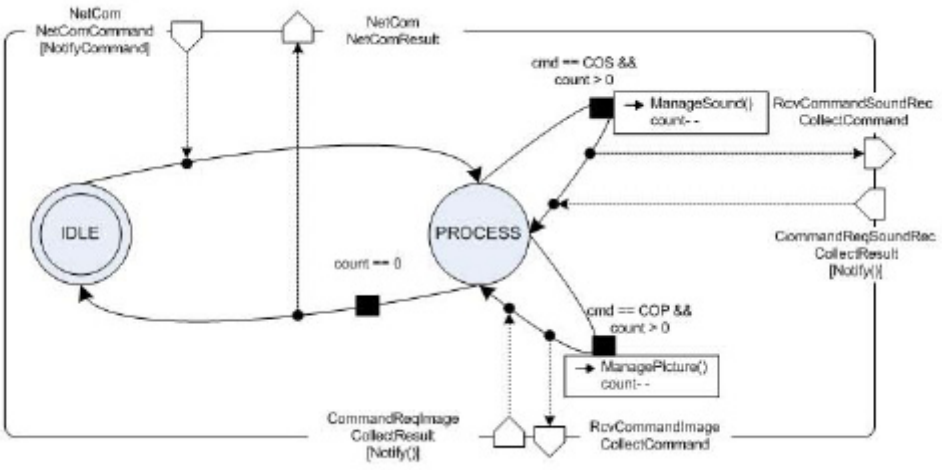

Figure 9.4: Collect object.

XML

While collecting, a XML file is created containing the files information corresponding to the current collect. The produced XML file just consists on a list of element identified by the type of collect. So, for now, only SOUND and PICTURE element are added in the file, the information inscribes in the XML file is their files name. This XML file will be used by the Web Server to display the data requested.

9.2.3 ImageObserver and SoundRec objects

Both objects are only briefly described here because they are an adaptation of a sample provided by Sony. They have been used and modified to show an application of the MFART protocol.

Aibo ERS-210A is equipped with a digital camera. The system which manages the camera sends a message with the capture from the camera each times it is requested. When the object is in *CAPTURE* state, the data from the camera are saved in a BMP file using the encoder provided by Sony. In our adaptation, only a message from the collect object can change the state to *CAPTURE* and then permit to take a picture. The figure 9.5 shows the object's operations.

Aibo ERS-210A is also equipped with a microphone. The system which manages the microphone sends a message with the sound recorded every 32 ms. When the object is in *START* state, the recorded sound is saved in a buffer while it is not full. Then, the buffer is saved in a WAV file. The defined size of this buffer will determine the duration of the record[1]. Only a message from the collect object can change the state to *START*and then permit to record a sound. The figure 9.6 shows the object's operations.

9.3 Encountered problems

During our development, we often were in front of a technical problem coming from the OPEN-R SDK technology. With ERS-210A, it is not possible to use the remote control to supervise running time like with ERS-7. At running time, it was only possible to

[1] 32ms takes 2048 bytes in the disk, so to record a sound of 16 seconds, by sample, it is necessary to define a buffer of 1048565 bytes

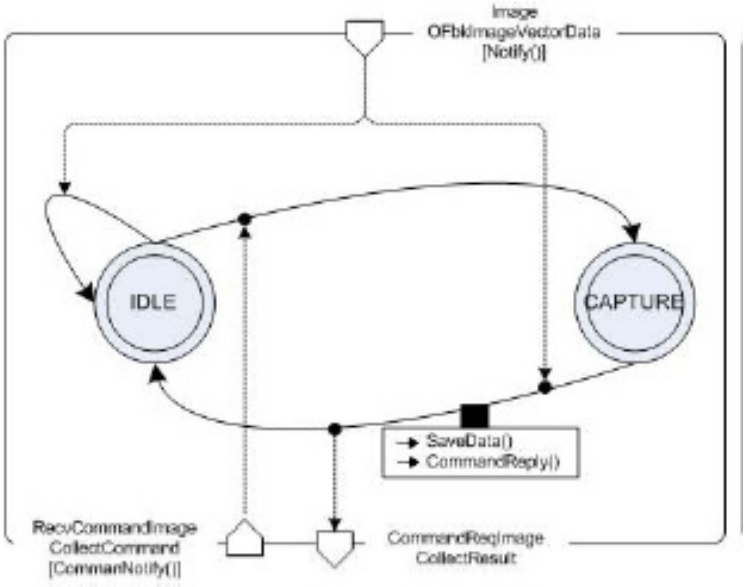

Figure 9.5: ImageObserver object.

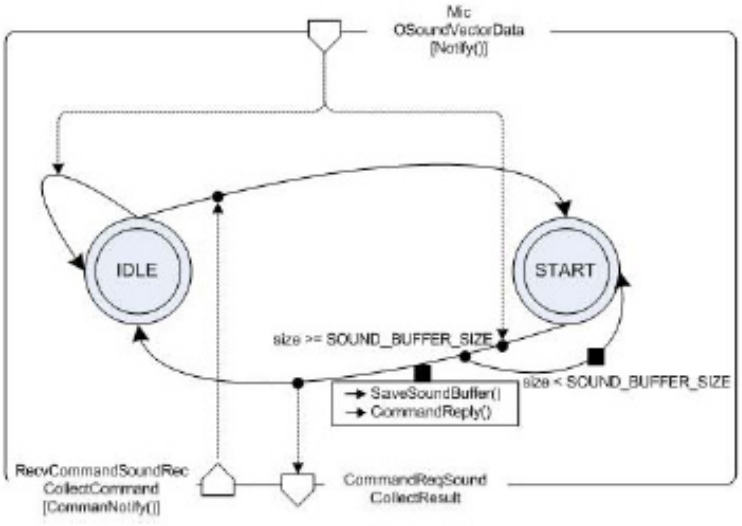

Figure 9.6: SoundRec object.

observe a console connected with ERS-210A via a telnet session. The *PowerMonitor* object provided by Sony shows all messages that we want on this console. But when a bug occurs, the system crashes, *PowerMonitor* is shutdown and can not display messages on the console. By consequence, it is difficult to debug using `println()` as usual. There is a debugging scripts, but it is not completely correct and it often provides poor information.

Chapter 10

Web Server Implementation

This chapter will talk about the implementation of the *Web Server*. So first of all, we will justify our technology choices for the selected language and also for our web server were the web site will run. Second, we will review the most important characteristics of the WS and third, we will finish with a small digression on the problems we met during its implementation.

10.1 Technology justification

10.1.1 Java programming language

We opted for Java, a compiled language for some major reasons. First, programming languages are faster to execute. The biggest particularity of our system is the frequency of its *hints*. Most of the time, only few connections will come to it, but when any catastrophe happens, a huge number of requests should be sent and many files should be received at the same time. And above all, recall that the biggest goal of the WS imposes that quick answers must be given to rescuers. Fast and efficient request processing was so primordial for our application.

Second, Java offers one of the most impressive set of packages as `java.util` for text processing and `java.net` for network connections. In addition to official SDK releases, many programmers distribute freely their personal upgrades. Among other facilities provided by Java that were used here, we note network connections management, (XML)

file processing and schedulers but most of all, its *error stack* and a remarkable parallel process execution.

Third, a strongly typed languages was useful for inner data structuring since the configuration files of the LS gather many different type of information (IP, list of Aibos, etc.). Furthermore, strongly typed language often go with easy code-debugging.

Fourth, it was also necessary to make the source code as clear as possible because our project will be carried on by other persons. To this end, JSP code help to not overuse boring `out.print()` for simple web page letting them easy to understand and Servlet classes structure more complex processing.

Last, we wanted to make our web site highly portable. Java is clearly platform independent (even if the GUI objects management has sometimes some small platform-dependent particularities) and thus answered to our most critical requirements.

10.1.2 Apache Tomcat web server

After the programming technology, the next step was to choose a web server. Nowadays, several of them supports the Servlet/JSP technology. But the choice was a bit made in advance. In term of reliability, performance and stability, Apache precedes all its rivals excepting maybe the Zeus server[1]. Unfortunately, its cost did not allow us to test it as well as the IIS server. Even if Apache seemed to be the only possible choice, it was nevertheless a thought decision looking to its assets. In addition to its performance, it is also easy to set up and to maintain.

Coupled with Apache, the Tomcat extension provides a powerful Servlet Container. Besides, last releases enable to modify and dynamically load servlets while the server is running. Most of all, Apache is running as well on Windows as on Linux platforms and is born from the open source community.

Apache servers have been tried and are running all over the world for many years and once again, in a goal of portability, Apache sure was the best choice, especially if the web server will run over Linux.

[1] see section 5.3.3

10.2 Web site's major functionalities

The following section will cross over the most important possibilities that the web site offers. Behind these functionalities, we will explain the underlying operating of the *Web Server* developed to support rescue teams.

10.2.1 Information collecting requests

The original goal of the web site was allowing rescuers to see the Aibos connected to the network, asking them to do some work and last gathering the results of this. The management of collecting requests represents the fundamental part of the web site.

In a previous chapter, we talked about the `config.xml` file and about its use[2]. In order to correctly display the list of Aibos registered near a *Local Server*, the WS need to have stored all config files of the known servers[3]. Like this, the display of this list for rescuers does not need previous requests to the local servers and it also allows to know any information the administrator of each LS has filled in the file (particulary important for specialties and locations parameters).

When a rescuer selects one or more Aibo(s) on the web site and want them to explore their environment, the WS create for each Aibo a **NOTIFY** message divided in two parts. The first part is the name of the chosen Aibo, and the second is the request itself beginning with a **BEGIN** notification and composed by either a **CMD_COP** or **CMD_COS** command for Picture taking or Sound recording. This command will also contain every technical arguments separated by semicolons. We chose to send one **NOTIFY** message by Aibo to use very small HTTP data packets and like that, increase the notification rapidity and exploit at maximum the parallel easiness of the two servers (LS and WS).

Afterwards, when all notifications have been dispatched, the WS sends back the user to the display page. Unfortunately, until now, no automatic update of the web page is available, rescuers must do it by themselves, but in a worry of leaving the WS as free as possible, we wonder if automatic actualization will not pick up to much connection time and availability.

[2] see section 8.2.1

[3] we will deal with this operation in section 10.2.2

On this web page, people can look at all files forwarded by Aibos grouped by their LS location and by type. The web site keeps them in its database and the administrator is free to manage them as he wishes. They are organized by LS and referenced by the name of their *explorer* and the creation date and time.

10.2.2 Subscribing and updating Local Servers

In order to keep the list of all local servers and their Aibos, two mechanisms were developed. The first is a subscribing page to register LSs and the second is an update process when known LSs make some change in their local system (number or names of Aibos, location, etc.).

First, when someone possessing an Aibo want to join the network, she or he has to connect to the site. For the purpose to ensure that it effectively concerns a LS, the WS asks the local IP and port of the LS. In fact, the WS sends a **GETCONFIG** message and wait for the answer. If it obtains a well-formed `config.xml` file, it then asks the person to fill in a subscribing form with a login, password and some personal information destined to rescue teams.

Second, with the object of keeping the WS database up-to-date, LS owners must upload their config files every time they make a single change in it. They have to connect to the update web page of the WS and ask it to download the file. The WS uses here again a **GETCONFIG** message and wait for the answer as it does for subscribing calls. It is crucial to maintain a high-level of coherence between the LSs and the WS database since disasters can happen at any time. In the improvements chapter[4], we will discuss about the possibility of an automatic update.

10.2.3 Subscribing rescuers

Since only rescuers can send requests to Aibos, a *login-password* system was required. Also, we had to create a special management of these logins because the administrator of the WS must be sure that the person asking a registration is effectively a member of a rescue team. After analyzing different possibilities, we decided to create a *demand-to-register* form. If someone wants to subscribe as a rescuer, he first fills an application

[4] see section 11.3

form that will be stored on the WS database in a dedicated XML file. The administrator is in charge of looking at these requisitions and validate the ones that are legitimate. The admin, and only him, can ensure (maybe by other means than the web site only, for instance contacting directly the applicant) that no ill-disposed person obtains unwanted priorities.

When the admin want to validate a submission, he selects the corresponding person on its list and the web site will automatically send an electronic registration proposal to the applicant. A *nonce*[5] is sent to the applicant and is used to access the registration page. This is also used to add one more security level on our system.

But, why all these precautions? Rescuers are the only persons authorized to send requests because some jokers could find funny to make Aibos making collect jobs for entertainment. These kind of excessive calls could cause troubles in term of privacy. That is also why we think about allowing only rescuers to look through the collected files in the future. Furthermore, we particulary want to avoid every possibility of system hacking.

10.2.4 Actualization of Aibos statuses

In spite of all its assets, Aibo has one notable problem: its battery. Our Aibos were working during one hour and a half at maximum. So, during a day, Aibos connect and disconnect to their LS frequently and the coherence with the Aibos displayed on the web site could be damaged. We had to create a way to increase its coherence and in that aim, the WS frequently sends **GETSTATUS** messages.

These messages are used to ask to each LS to forward presents statuses of its Aibos. Thus, the web tends to stay as coherent as possible. Note that the message frequency is easy to set up. The critical point here resides in the state of the version of the LS config file on the WS. If it is not up-to-date, some status coherence problems will occur since the LS answers with a list of integers representing the statuses of its Aibos, and this lists is ordered following the `config.xml` Aibo list. Here again, a systematic update of the config file would strengthen the general system.

[5] number used only once

10.3 Encountered problems

This last section of the current chapter will briefly recall the most tricky problems we met during the WS design, development and implementation. We will not mention every single detail, but rather, we will talk about the three major issues we encountered.

10.3.1 Specification lacks

At the very beginning, the project was explained on two single small pages only filled with two drawings and several vague explanations. Coupled with communication problems due to the language difference, we had some difficulties to figure what exactly was our task. Little by little, we cleared the project scope and we obtained some precisions over the work to accomplish.

In fact, we had to start a project totally from scratch and began by analyzing precisely the job. As in a *real* project, we made a first report with a scope analysis. After a discussion with Professor Tadokoro who validated it, we started the design and implementation of the system.

As inexperienced programmers, we stumbled on many beginner's issues and the system architecture evolved many times, causing many implementation changes and tests. But, even if they are not perfect, the protocol and the application created are working and they win in portability and evolving capabilities.

10.3.2 Security questions

The security question was the most important one for the system. We had to think of an original manner to manage subscribing of rescuers and try to secure the network of *Local Servers*. Once again, the lack of experience made us taking many times to opt for one solution rather than another. The system is easily improvable by restraining the data access to rescue teams and the actual login system could be enlarged to meet this requirement.

Besides, we thought about using secured connections for every data transferred from and to the WS, but we were short with the time. The implementation lets one free to

add this upgrade since SSL[6] connections exist in Java.

10.3.3 Web programming from scratch

Our formation did not prepare us to deal with web technologies. CGI, ASP, PHP,..., here are too many acronyms for novices. Starting a web application from zero asked us to have a shake-up between all these technologies before. Not only, we had to look through new languages, but also, we had to familiarize with web servers. The time necessary to understand and to become accustomed to this new world shortened the development time. The proliferation of available technologies made the choice difficult. We wanted to be sure to choose the most promising technology, so we took our time to look at the pros and cons of each of them.

The computer science knows a new *boom* since holding a web site became so popular and spread. Behind the at-first-sight hard labour to see more clearly in this web world, it gives us the opportunity to take a glance to this growing part of the computer science: web development.

[6]Secure Socket Layer

Chapter 11

Protocol and application improvements

11.1 Architecture

The protocol that we developed reveals a mechanism to manage the communication with Aibo which could be improved.

11.1.1 Current architecture

For the moment, when a collect request is processing with a predefined Aibo, the *Local Server* has to lock this Aibo, changing its state to *WORKING*, to prevent all other request for this one. This lock is released when the request processing is completely finished. So, the LS has to enqueue the requests for Aibos and resend them when Aibo is ready.

That constitutes a weakness, in architectural terms, because the *Local Server* and Aibo are not completely independent9.2.1. Treatments in LS become heavier because of the process time in the Aibo's *NetCom* object.

When Aibo is launched, the server starts and listen to a predefined port. When a client's request is intercepted by the listening port, the remote address is aught to open a communication channel to send the request results. These one could be rough text or file according to the initial request.

The server is listening again when the client's request is completely done. If the LS has to forward another request to this Aibo, this one will have to wait on the LS because of Aibo's state. Without this state mechanism, the LS thread would have been blocked until TCP connection timeout was finished.

11.1.2 Solution

To optimize our architecture, we have defined a new system of communication between both hosts. The first part of this improvement concerns the communication stream with Aibo, and the second one modifies the handling of Aibo by the LS. This improvement has not been developed because we were short of time, but all analysis points are described here below.

Communication stream with Aibo

In place of an unique channel of communication between Aibo and the *Local Server*, we propose to use the FTP's principle. FTP uses two network communications, one to exchange commands, and the other one to send data.

In the *NetCom* object, it will be the same, a first connection will be used to receive commands from LS and the other one to send results. With this configuration, the server is listening at all times. Each time it receives a request, *NetCom* processes it and, according to its availability, sends a *success* or *failure* message to the LS. If Aibo was not busy with another request, the request is processed as explained in section 9.2.1.

Four important steps have to be followed to implement this improvement:

- Test Aibo's state when a client's request is coming
- Send a success or failure message to the *Local Server*
- Save the remote IP address, contained in the client's request, in a variable
- Open a new TCP connection with the client's IP address to send the request's results

Aibo handling by Local Server

Given this previous improvement, the LS could be consequently simplified. It can now skip the entire monopolization step. In this step, the LS had to test Aibo's availability, according to this, change Aibo's status to *WORKING*. After completion of a request, release monopolization changing Aibo's status to *READY*. In case Aibo was busy, the request was packed in a **Message**[1] and added in the *Waiting Queue*. With the modification, each request can be directly forwarded to Aibo without any test. But now, when a request is forwarded to Aibo, the LS has to wait for notification. In case of success, it waits for the results, otherwise, the request is packed in a **Message** and added in the *Waiting Queue*.

Here are the steps for implementation:

- Remove monopolization procedure
- Wait for notification of request forwarding from Aibo
- Change status displays and Aibo's state test on Graphical User Interface
- Change configuration synchronization of Aibo's state between the *Local Server* and the *Web Server*

11.2 Recommendations for streaming implementation

11.2.1 Context and origins of the improvement

The new architecture proposed in the previous section made us thinking about a larger scale improvement. It is possible to use Aibo as a real-time explorer. Streaming technologies works with architecture close to the one proposed above, so we thought that the new system should contain a similar possibility.

11.2.2 Preliminary remarks

Before thinking to the implementation, we have to check if the existing system can support such a modification without large revisions. The architecture simplification will

[1] see section 8.2.2

ease the addition of streaming. The simplified LS will not change since requests are going more transparent for it. A new pair of messages will maybe appear for starting and closing streaming ports[2]. The two other hosts need more thoughts.

Apache web servers have some problems to correctly handle streaming diffusion. But a new open source implementation of the Apple Streaming Server called the Darwin Streaming Server[36] has been released and made available the source code. On the WS side, the implementation of the video display will take some time but the underlying system will be very small.

Sony developed an environment to look at Aibo's functioning and to display a real-time view of what it *sees* with its camera. This tool, the *Remote Framework*, is only available over the ERS-7 models and uses a protocol of Sony. As only the compiled code is available, it is impossible to find implementation details on their *RFW* system. However, for the 2X0 models, we found an alternative program created by Aibo fans named *AiboScope* that copy parts of the *RFW*. The source code with some explanations are available on [30].

11.2.3 Protocol choices

The chapter 6 gave an overview of the most known streaming protocols. Here, we have the choice between taking existing protocols, starting from scratch or mixing the two approaches. There is no best way, but here is our solution.

To be sure to avoid any problems, we advise to use SDP messages for sessions handshaking. Encoding formats do not have to be negotiated in our case. These messages are small and contain no layout or file organization details. There are small text files, so simple to create on Aibo.

For transport and control protocols, RTP and RTSP are the most used ones and they let free to use TCP or UDP. They are totally independent and works well with SDP. This is the usual combination took for streaming implementation.

Using the distributed code of the Darwin server which works with SDP, RTSP and RTP, the WS upgrade will take a bit of time, but the existing code eases the tasks. But, because the only available code is the *AiboScope* program for Aibo, it needs several

[2] see section 11.2.4

modifications to implement these protocols and to fit in our system.

11.2.4 Implementation recommendations

We have some last recommendations. It is not necessary to totally implement these protocols on Aibo. Since it will only display live video, only a few of the RTSP messages have to be implemented.

The LS maybe need two new protocol messages for *begin-teardown* commands. Note that this modification is not obligatory because it acts as a *forwarder* and let the communication channel open until Aibo has finished.

The whole system will gain more power if existing protocols are implemented, even if their scopes surpass the present use. It is also possible to create a simple RTSP-like protocol which is in charge for initiating and controlling the stream flow simply running on UDP or TCP. This is the easiest way for Aibo but the work on the WS side will be even more difficult because of the abstraction layer of the Servlet Container4.6.

11.3 Web site improvements

11.3.1 XML header file analyzing

As explained in section 7.2, each answer from Aibo is preceded by a XML file. Currently, this file is small and regroups only little information. But, we decided to add it before each response to anticipate future uses of our system. We knew that the IRS lab wanted to connect a lot of other devices to the domestic network such as a smoke detector. These machines create only text data and a special file has to gather them. Our protocol foresees the future development possibilities of the local network and the header file will be completed with these data.

Using XML was a major asset because file parsing is easy. Furthermore, different version of the DTD could be used to customize each header file depending on the network configuration.

However, until now, this file is only stored on the server side and used for a technical purpose. But, the IRS lab is working on a common XML-based specification for special *rescue data files*. The IRS and other labs and universities in Japan are working on the

XGD[3]-specification. It defines a common grammar for XML files used to exchange geographical and other general-purpose data. This specification was not completely finished when we ended our work there, so we let our header file almost empty anticipating its evolution following XGD.

11.3.2 Pause notification

In section 10.2.2 and 10.2.4, we raised two limits of the web server. If the configuration file of the local server is modified, its owner has to connect to the web server to upload its new file. If he forgets, the WS version of the `config.xml` will be different from the real one, with the problems that this involves. We thought about a mechanism to be sure that the person in charge of the LS never forgets to actualize its file on the WS.

We thought about a *pause* and a *ready* notification. When the file has to be modified, the LS will send a *pause* notification to indicate the WS it will be unavailable for a moment. When it has finished, it sends a *ready* message and asks to upload its config file. These two notifications will decrease the coherence lacks risks between the LS and the WS.

11.3.3 Web site design

Our job was to create a complete working system. Unfortunately, we did not have much time to embellish the graphical content of the web site. It is working but it is not really beautiful and many design work still remain. We decided to not worry too much about the graphical attraction to let us concentrate on the underlying system which seemed more important to us.

11.4 Security

Each system exchanging data between remote hosts raises security questions. As collect's requests are asked from the web site by a rescue team, we have used a session system[4]

[3] eXtendable Geographical Database System. Until now, the documentation is only available in Japanese.

[4] see section 10.2.3

to protect this operation. Thanks to that we are sure that no unregistered users can manipulate Aibo. It is necessary, because Aibos have to be available and with charged battery for rescue operation and for privacy reasons.

At this moment, results are displayed on a public web page and could be seen by everybody. But in case of real rescue operation, those results will be confidential and visible only by rescuer. Problem can occur if hackers try to intercept data on the network or on the *Web Server* memory. An interesting improvement to prevent this is to use encryption technology to protect data from Aibo to *Web Server*. Some encryption algorithm like *RSA* or *TripleDES* are effective to protect pictures or sounds files.

Furthermore, it will be interesting to improve security for personal information transmission. Because security is the most important priority on this kind of system, before effectively running, the application must be completed with a security module at least for login and password transmission. Because we used Java, this additional process will not be too difficult to set up.

Chapter 12

Conclusion

In the present work, we created a communication protocol between Aibo and a web site. This protocol allows people to connect Aibos on a local network and make them available for external requests. Rescue teams can use the web site to see connected dogs and ask them to collect information.

First, we created a *Local Server* managing a set of Aibos and acting as a middleware host. Its light and efficient implementation enables it to manage many requests using a Waiting Queue. Moreover, this server is completely independent of the crossing data thanks to the message encapsulation system.

Second, we developed a special OPEN-R object acting as a middleware too. The *NetCom* object manages the incoming requests, transmits them to the *Collect* object that deals with the data gathering, packs up the results and sends them to the LS.

Lastly, the *Web Server* displays all registered Aibos that are currently switched on. Rescuers can connect to it, ask some Aibos to collect information and check the results on a web page.

Developed in a reduced environment, this project offers an analysis which can be generalized for large scale disasters. The communication principle of Aibo using wireless technology could be exactly the same with rescue robots progressing in a hostile area. The difference is that our protocol uses Aibo, a fragile and restricted robot, and is adapted for usage in a known small room.

We see our system as the basis of a new striking complete communication system for

rescue teams. Beyond the experimental use, we hope that we helped to create a working protocol and application that will be used as a ground work for future developments. With the explosion of electronic devices in households, our application let the local server administator free to add as many entities he wants. Furthermore, our message system is high-level enough to impose nothing on the implementation.

Our system has been composed very simply, so adding new messages or changing details of existing ones is a peace of cake. We always kept in mind that we only developed the foundation and tried to create a completely modular system.

Bibliography

[1] C.E.BROWN, *ASP vs. PHP - Which one is right for you*, http://www.pointafter.com/Archives/nl0203.htm, 1999

[2] M.BROWN, *IIS vs. Apache, Looking Beyond the Rhetoric*, www.serverwatch.com/tutorial/, 2003

[3] B.ALBAHARI, *A Comparative Overview of C#*, http://genamics.com/developer/csharp_comparative.htm, 2000

[4] O.CAUDRON, *Build Competitive Edge Through Innovation: Java contre .Net*, InterSystems Symposiums, 2005

[5] E.COSTE-MANIERELA, *Robotique applique la chirurgie*, http://www.futura-sciences.com/comprendre/d/dossier152-1.php, futura-sciences, 2002

[6] D.FERGUSON, *Zeus Technology: Comparing High Performance Web Servers*, Zeus technology, 2002

[7] M.HALL, *Core Servlets and JavaServer Pages*, Prentice Hall & Sun Microsystems Press, 2000

[8] M.HANDLEY, V.JACOBSON, *RFC 2327: SDP: Session Description Protocol*, MMUSIC Group, Network Working Group, 1998

[9] C.HENG, *Comparing PHP scripting with CGI scripting using Perl*, http://www.thesitewizard.com/archive/phpvscgi.shtml

[10] J.HOBBS, *Tcl 8.4 Overview*, 9^{th} Annual Tcl/Tk Conference, Active State Corporation, 2002

[11] S.HULL, *PHP and ASP.NET Go Head-to-Head*, http://www.oracle.com/technology/pub/articles/hull_asp.html

[12] E.E.KIM, *CGI Unleashed*, chapters 1-7;21-25, http://docs.rinet.ru:8083/UCGI/, Sams.net Publishing, 1996

[13] MACROMEDIA, *ColdFusion MX7 Getting Started Experience Tutorial*, Macromedia, 2005

[14] , D.MARKATOS, *Introduction to ADO.NET*, http://www.sitepoint.com/article/introduction-ado-net, 2003

[15] A.MICHALAS, V. ZOI, N.SOTIROPOULOS, N.MITROU, V.LOUMOS and E.KAYAFAS, *A Comparison of Multimedia Application Development Platforms towards the Object Web*, National Technical University of Athens, 2000

[16] OPEN-R Programming Special Interest Group, *Introduction to OPEN-R programming*, Sony Corporation Entertainment Robot Company, impress corporation, Japan, 2002

[17] A.POSSOZ, *Cours TCL/Tk: EPFL 20-23/05/97*, Service Informatique Central-Section Logiciels, Ecole Polytechnique Fédérale de Lausanne, 2005

[18] C.PREIMESBERGER, C.AREHART, *Why Java Should Not Temper ColdFusionML Talents*, http://www.oetrends.com/news.php?action=view_record&idnum=232, 2003

[19] F.PRIAM & J.Steffe, *Apprendre PHP*, ENITA Bordeaux-UF Informatique, 2002

[20] J.ROSENBERG, H.SCHULZRINNE, G.CAMARILLO, A.JOHNSTON, J.PETERSON, R.SPARKS, M.HANDLEY, E.SCHOOLER *RFC 3261: SIP: Session Initiation Protocol*, Network Working Group, 2002

[21] H.SCHULZRINNE, S.CASNER, R.FREDERICK, V.JACOBSON, *RFC 1889: RTP: A Transport Protocol for Real-Time Applications*, Network Working Group, 1996

[22] H.SCHULZRINNE, A.RAO, R.LANPHIER, *RFC 2326: Real Time Streaming Protocol (RTSP)*, Network Working Group, 1998

[23] M.SEMILOF, *Kicking the Windows habit: Apache vs. IIS*, http://searchwin2000.techtarget.com/originalContent/0,289142,sid1_gci833798,00.html, 2002

[24] F.SERRA, J-C.BAILLIE, *Aibo programming using OPEN-R SDK - Tutorial*, http://uei.ensta.fr/baillie/eng/openr_tutorial.html, Ecole Nationale Superieure de Techniques Avancees, France, 2003

[25] SUN MICROSYSTEMS, *Comparing JavaServer Pages Technology and Microsoft Active Server Pages : An Analysis of Functionality*, Sun Microsystems Press, 1999

[26] A.TERZIS, B.BRADEN, S.VINCENT, L.ZHANG, *RFC 2205: Resource ReSerVation Protocol (RSVP - Version 1 Functional Specification*, Network Working Group, 1997

[27] G.van ROSSUM, *Python Tutorial Release 2.4.1*, Fred L. Drake, Jr. editor, 2005

[28] J.WILSON, *(Macromedia) Cold Fusion: A Brief Overview*, Vanderbilt University, 2002

[29] *Aibo Life web site*, http://www.aibo-life.org

[30] *Aibo Fans programming web site*, http://www.aibohack.com

[31] *Answers.com*, http://wwww.answers.com/mbone.htm

[32] *AWK programming language*, www.thefreedictionary.com/AWK+programming+ language.htm

[33] *Center for Robot-Assisted Search and Rescue*, http://crasar.csee.usf.edu/MainFiles/index.asp, University of South Florida

[34] *Choosing A Scripting Language For Your Web Site*, http://www.comparewebhosts.com/Article.asp?ArtId=7

[35] *Active Web Sites and Comparison of Scripting Languages*, http://training.gbdirect.co.uk/courses/php/comparison_php_versus_perl_vs_asp_jsp_vs_vbscript_web_scripting.html

[36] *Darwin Streaming Server*, http://developer.apple.com/darwin/projects/streaming/

[37] *FreeBSB Project*, http://www.freebsb.org

[38] *GraphComp web site*, www.graphcomp.com

[39] *How JSP compares with ASP*, http://www.daysite.net/programming/jsp.htm

[40] *Intelligent System. Thinking Technology*, http://www.precarn.ca/home/index_en.html

[41] *International Rescue System Institute*, www.rescuesystem.org, Japan, 2002

[42] *L'Aibo Qu'est-ce que c'est?*, http://www.vieartificielle.com/article/index.php?action=article&id=195, vieartificielle.com, 2004

[43] *L'histoire Aibo*, http://www.vieartificielle.com/article/index.php?action=article&id=197, vieartificielle.com, 2004

[44] *Macromedia France FAQ*, http://www.macromedia.fr/FAQ

[45] *Microsoft Developer Network*, http://msdn.microsoft.com

[46] *Microsoft Support web site*, http://support.microsoft.com/

[47] *Official Aibo web site*, http://www.sony.net/Products/aibo/, Sony corporation, 2005

[48] *Official Apache web site*, http://www.apache.org

[49] *Official PHP web site*, http://www.php.net

[50] *Official Robocup Rescue web site*, http://www.rescuesystem.org/robocuprescue/

[51] *Official Zeus web site*, http://www.zeus.com

[52] *Open-R web site* http://openr.aibo.com/, Sony Corporation, 1995

[53] *Remote Operation with Supervised Autonomy*, http://iit-iti.nrc-cnrc.gc.ca/successes-reussites/rosa-mdr_e.html, Institute for Information Technology, Canada, 2001

[54] *RoboCup*, http://www.robocup.org/

[55] *RoboCup 2005 OSAKA*, http://www.robocup2005.org/home/default.aspx, Japan, 2005

[56] *RoboCup Cornell*, http://robocup.mae.cornell.edu, Cornell University, United States of America, 1998

[57] *RoboCup University Laval*, http://www.robocuplaval.com, University of Laval, Quebec, 2004

[58] *RSVP Protocol Overview*, http://www.ietf.org/html.charters/sip-charter.html

[59] *Simon's Web Site*, http://www.simonrobinson.com/DotNET/DotNET.aspx

[60] *Sun Web Site*, http://www.sun.com

[61] *Synchronized Multimedia Integration Language 1.0 Specification*, http://www.w3.org/TR/REC-smil/, 1998

[62] *Tcl/Tk tutorial*, http://www.techiwarehouse.com/Tcl_Tk/Tcl_Tk_Tutorial.html

[63] *Team Canuck*, http://www.cs.ualberta.ca/r̃obotics/robocup, University of Alberta, Canada, 2003

[64] *Wikipedia Encyclopedia*, http://www.wikipedia.com/mbone